Vítor Rodrigues

Semantics-based Program Verification: an Abstract Interpretation Approach

Doctoral Program in Computer Science
of the Universities of Minho, Aveiro and Porto

Dezembro de 2012

Vítor Rodrigues

Semantics-based Program Verification: an Abstract Interpretation Approach

Thesis submitted to Faculty of Sciences of the University of Porto
for the Doctor Degree in Computer Science within the Joint Doctoral Program in
Computer Science of the Universities of Minho, Aveiro and Porto

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
Dezembro de 2012

To my Family

Acknowledgments

First of all, I would like to give thanks to my supervisors Simão Melo de Sousa and Mário Florido for all the help. Our objective to extend the use of *worst-case execution time* (WCET) analysis based on abstract interpretation to the verification of mobile embedded real-time software has revealed itself a great and unexplored challenge, from both formal and technical points of view, and the fact that we have published one paper for each particular branch of research was of great satisfaction for the three of us.

I am truly grateful for the technical contributions given by João Pedro Pedroso in the verification of linear programming and for the dedication of Benny Akesson in helping us finding an efficient use timing abstractions in the analysis of multicore architectures with shared resources. I am also grateful to David Pereira and Michael Jabobs for their patience and frutiful discussions.

The thesis would not have been the same without the intellectual environment provided by Compiler Design Lab, at the University of Saarbrücken, Germany. There, I had the honor to meet Prof. Reinhard Wilhelm and his team of PhD students and colleagues, in particular Jan Reineke. My stay in Saarbrücken for the period of three months was the starting point for the design and implementation of our WCET analysis in multicore architectures.

This dissertation is the result of my own work and includes nothing which is the direct outcome of work done by others.

I hereby declare that this dissertation is not substantially the same as any I have submitted for a degree or diploma or any other qualification at any other university. I further state that no part of my dissertation has already been, or is being currently submitted for any such degree, diploma or other qualification.

Abstract

Modern real-time systems demand *safe* determination of bounds on the execution times of programs. To this purpose, program execution for all possible combinations of input values is impracticable. Alternatively, *static analysis* methods provide sound and efficient mechanisms for determining execution time bounds, regardless of knowledge on input data. We present a calculational and compositional development of a functional static analyzer using the theory of *abstract interpretation*. A particular instantiation of our data flow framework was created for the *static analysis* of the *worst-case execution time* (WCET) of a program. The entire system is implemented in Haskell, with the objective to take advantages of the declarative features of the language for a simpler and more robust specification of the underlying concepts.

The target platform for program analysis is the ARM9 microprocessor platform, commonly used in many embedded systems. The *verification* of embedded programs is performed directly in the compiled assembly code and uses the WCET as the safety parameter. We reuse the notion of *certificate* in the context of Abstract-Carrying Code (ACC) and extend this framework with a verification mechanism for *linear programming* (LP) using the certifying properties of *duality theory*. On the other hand, our objective is also to provide some degree of WCET verification at source-code level. Since the information about the WCET can only be computed at machine-code level, we have defined a *back-annotation* mechanism that is based in compiler debug information and was designed to be compiler-independent and used in a transparent and modular way.

The WCET analysis is extended to multicore architectures with shared resources by applying the latency-rate ($\mathcal{LR}$) server model to obtain sound upper-bounds of execution times of programs sharing resources. The soundness of the approach is proven with respect to a calculational fixpoint semantics for multicores, which is able to express all possible sequential ways of accessing shared resources. Experimental results show that the loss in precision introduced by the $\mathcal{LR}$ model is about 10% on average and is fairly compensated by the gain in analysis time, which is above 99%.

Resumo

Os sistemas de tempo-real atuais baseiam-se em estimativas sobre o tempo de execução dos programas. Para este fim, a execução de todas as possíveis combinações de dados de entrada do programa é impraticável. Em alternativa, os métodos de *análise estática* proporcionam mecanismos corretos e eficientes para determinar estimativas sobre o tempo de execução, independentemente dos dados de entrada. Nesta dissertação, é apresentado um analisador estático, concebido através de um método de cálculo composicional baseado em *interpretação abstrata*. A análise estática do *worst-case execution time* (WCET) de um program é definida como uma instanciação particular da nossa plataforma de análise de "dataflow". O protótipo foi desenvolvido usando a linguagem Haskell com o objetivo de tirar partido das suas caraterísticas declarativas, para assim obter uma especificação mais simples e robusta dos conceitos envolvidos.

A plataforma de hardware escolhida para a análise de programas é a plataforma ARM9, a qual é muito comum em sistemas embebidos. A *verificação* de programas embebidos é efetuada diretamente no código compilado "assembly" e usa o WCET como parâmetro de segurança. É reutilizada a noção de *certificado* no contexto da plataforma "Abstract-Carrying Code" (ACC), a qual é estendida com um mecanismo de verificação para *programação linear* (PL) baseado na *teoria dual* associada a PL. É também promovido o uso do WCET para verificação dos tempos de execução diretamente no código fonte. Para este efeito, é definido um mecanismo de "back-annotation" que se baseia na informação de "debug" produzida pelo compilador.

A análise do WCET suporta também arquiteturas "multicore" ao aplicar o modelo de "latency-rate ($\mathcal{LR}$) servers", o qual permite obter estimativas corretas sobre o tempo de execução de programas que partilham recursos. A prova de correção desta abordagem é feita em relação à semântica de pontos-fixos para "multicores", a qual é capaz de expressar todos os modos nos quais os recursos partilhados podem eventualmente ser acedidos. Resultados experimentais demonstram que a perda de precisão introduzida pelo modelo $\mathcal{LR}$ é cerca de 10% em média, mas é em grande medida compensada pelo ganho no tempo de análise, o qual é superior a 99%.

Contents

Chapter 1

Introduction

The design of embedded real-time systems is directed by the timeliness criteria. By *timeliness* we mean that real-time programs, possibly running on multi-core embedded systems, have operational deadlines from system event to system response and must guarantee the response within strict time constraints. Timeliness evaluation is performed at system level and is defined as the capability of the system to assure that execution deadlines are meet at all times. Therefore, when the risk of failure, in terms of system response, may endanger human life or substantial economic values [75], the determination of upper bounds for the execution times of programs is a *safety* requirement in the design of embedded hard real-time systems.

The timeliness safety criteria [75] is most commonly specified by the *worst-case execution time* (WCET) of individual programs, i.e. the path inside a program that takes the longest time to execute. In the general case, the particular input data that causes the actual WCET is unknown. Therefore, the determination of the WCET throughout testing is an expensive process that cannot be proven correct for *any* possible run of the program. An alternative to this incomputable problem are the methods of *static analysis*, which are able to determine sound properties about programs at compile time. The purpose of *WCET analysis* is to provide *a priori* estimations of the worst execution time of a program without actually running it. Nevertheless, the methods of *static analysis* are still far from being widely adopted in industry environments, which work more with tailored methods for specific processors using, for example, *detailed analysis*.

Notwithstanding, embedded real-time systems often require adaptive configuration mechanisms, where the update of available application services, or even operating system services, may be required after deployment. Traditionally this is done using manual and heavyweight processes, specifically dedicated to a particular modification. However, automatic adaptation of real-time systems can only be achieved if the system design abandons its traditional monolithic and closed conception and allows itself to reconfigure. In such hypothetical environment, the WCET analysis must be complemented with a verification mechanism, whose task is to verify whether an WCET estimate is compliant with safety requirements.

At the foundation of highly precise WCET analysis is the theory of Abstract Interpretation (AI) combined with Integer Linear Programming (ILP) techniques [142]. The need for static analysis by abstract interpretation of machine code follows from the fact that modern embedded microprocessors have many specialized hardware features that strongly influence the execution time. For this reason, timing analysis cannot be performed at source code level, considering the substantial imprecision that constant upper bounds on execution times would introduce in the WCET estimate.

Therefore, precise WCET analysis must be made at hardware level and must consider each hardware component that can affect the execution times. For example, cache memories and processor pipelines strongly influence the execution time. When performing WCET analysis, we are mainly interested in a flow-sensitive, path-sensitive and context-sensitive analysis of the local execution times of every instruction inside the program. Fig. 1.1 identifies the three required static analyses: a *value analysis* that computes abstract instruction semantics; a *cache analysis* that computes abstract cache states; and a *pipeline analysis* that computes concrete execution times, based on the concrete timing model of the processor. On the other hand, the global *worst-case* execution time is calculated *a posteriori*, using the static information locally obtained for each program point by abstract interpretation. This second step is termed by *path analysis* and is performed using linear programming.

More precisely, abstract interpretation is used to derive an approximate and computable abstract semantics of the behavior of the hardware components that affect the timing behavior of a program running on a particular microprocessor. As Fig. 1.1 shows, these components are the concrete (standard) instruction semantics of the processor's instruction set, the particular cache replacement policy of a cache memory and the timing model of the microprocessor pipeline architecture. Secondly, integer linear programming calculates the worst-case execution time of a program, in the scope of all its possible program paths, based on the "local" invariants statically computed by abstract interpretation for each program point along these paths. Our objective in developing an approach based on AI+ILP is to compute *verifiable* WCET estimates.

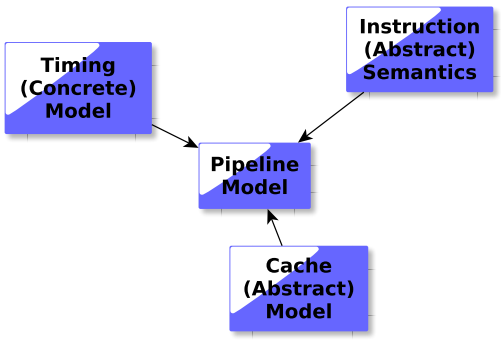


Figure 1.1: Overview of hardware components used in the abstract interpretation

In this context, verifiable WCET estimations as safety properties impose new challenges to the verification process, because of the nature of the techniques used to compute the WCET. The increasing dependency of the WCET on modern hardware mechanisms used to increase instruction throughput naturally increases the computational cost and complexity of WCET estimation. Considering the particular case of embedded systems, which typically have limited computing resources, the computational burden resulting from the integration of the complete WCET framework into the *trusted computing base* (TCB) of the embedded system would be unacceptable.

Therefore, to be effective, the verification mechanism should use lightweight and standalone methods to check if a given program satisfies a given safety specification in terms of execution time. In this thesis, the verification mechanism was designed to re-use the state-of-the-art methods of AI+ILP, although employing additional techniques to make such mechanism very time-efficient and low-resource consuming. In particular, the verification that the fixpoint of given a program is indeed its *least fixpoint* is made within the framework of Abstraction Carrying Code (ACC) for the abstract interpretation part, and the verification of the optimal solutions of the path analysis part is made using *dual theory* [88] applied to standard Linear Programming (LP), using the exact rational arithmetic of the simplex algorithm [66].

1.1 Abstract-Carrying Code

The central idea in Abstract-Carrying Code [9, 10, 11, 64] is the conceptual use of *least fixpoints* of programs in the abstract domain as *certificates*. These certificates carry properties about the dynamic behavior of a program that can be statically verifiable by abstract interpretation. Using fixpoint theory, the utility of these certificates is to avoid the re-computation of these properties for verification purposes. The prime benefit of the use of certificates in the context of embedded systems is the separation of the roles played by a code "supplier" and a code "consumer". The computational cost associated to the determination of the safety properties is shifted to the supplier side, where the certificate is generated. On the consumer side, the safety of the program actualization is based on a verification process that checks whether the received certificate, packed along with "untrusted" program, is compliant with the safety policy of the device.

To be effective, the *certificate checker* should be an automatic and stand-alone process and much more efficient than the *certificate generator*. However, several considerations concerning efficiency and precision of the process must be made. The method of AI+ILP is well-established to compute tight and precise WCET estimates. However, prominent WCET tools, such as AbsInt's aiT [2], rely on manual annotations of program flow information on the source code and perform separate static analysis for each hardware component that affect timing behavior. Although the separate analysis of each hardware component improves

efficiency, a verification mechanism based on such type of tools would be a heavy process that cannot be fully automatic.

Additionally, and besides the certificate checking time, also the size of the certificates will determine if the code actualization/verification in embedded systems can be performed stand-alone and in reasonable time [11].

1.2 Fixpoint Semantics

Program flow information about the source code typically includes the identification of infeasible program paths and maximal number of program iterations inside loops. The accuracy of this particular type of information is essential to avoid WCET overestimation [44]. Our proposal includes a method to automatically collect program flow information during fixpoint computations. This method is based on fixpoint algorithms based on *chaotic iteration strategies* [20, 33], which recursively traverse the program syntactical structure until stabilization is reached. Hence, chaotic iterations exactly follow the syntactic structure of the program, computing fixpoints as a syntax-direct denotational semantics would do. Hence, we propose the design of a *program flow analysis* that is automatically obtained by instrumenting the abstract domain used for *value analysis*.

Moreover, the design of a chaotic fixpoint algorithm provides a flow-sensitive analysis where the history of computation, particularly relevant for pipeline analysis, is taken into consideration. Additionally, path-sensitive analysis is performed by computing *backward abstract interpretations* of conditional branches. Support for interprocedural analysis is provided using the *functional approach* proposed in [128]. For these reasons, the static analyzer is able to compute the *value analysis* of registers and memory location simultaneously with the analysis of cache behavior and the analysis of the pipeline, using a generic and efficient fixpoint algorithm [115]. When using ACC, this feature is of great utility since it provides a one-pass traversal algorithm to check if the certificates actually behave as fixpoints.

Formally, the fixpoint algorithm is constructively defined as the *reflexive transitive closure* over input-output relations [28]. Each relation is associated with an instruction in the machine program, defined between two labelled program states. In order to correlate the shape of this fixpoint definition based on relational semantics, we apply the semantics projection mechanism proposed by Cousot [27], so that the data flow equations defined using denotational semantics can be complemented with the intensional information contained in the iteration strategy of a particular program. The corollary is a meta-semantic formalism that, besides providing an unified fixpoint form by means of algebraic relations, also supports program transformations to reduce the size of certificates and the verification time.

In our WCET toolchain, the static analysis plays the dominate role because the time

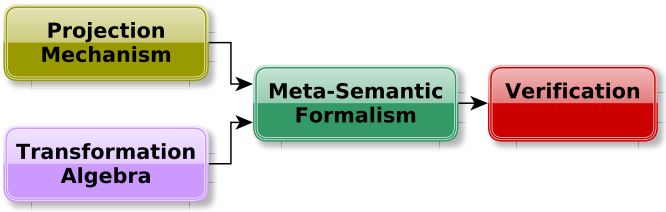


Figure 1.2: Interplay of the meta-semantic formalism with transformation and verification of programs supported by a semantics projection mechanism

necessary to generate the certificates with the local upper bounds for execution times is significantly greater than the time necessary to calculate the WCET using linear programming. However, the verification of the WCET cannot neglect the linear programming component because of the tools it uses. For example, the calculation of the WCET when using the *simplex method* [66] with constraints on the variables to be integers (ILP) is a NP-hard problem. Therefore, the inclusion of such tools in the embedded system for the purpose of verification is impracticable by design, due to the definition of the problem itself.

With the global objective in mind to restrict the *trusted computing base* (TCB) of an embedded system to one single pair of a code "supplier" and a code "consumer", we propose the inclusion of the WCET checking phase inside the ACC framework by relaxing the optimization problem of ILP to that of LP, by using the properties of duality theory and assuming that the coefficient matrix of the linear problem is totally unimodular [67] in order to preserve the integer semantics. The dual theory is based on the relation between the *primal* and *dual* solutions of the simplex methods. Using the mathematical properties of these two methods, the complexity of the linear optimization problem on the consumer side can be reduced from NP-hard (ILP) to polynomial time (LP), by the fact that checking mechanism only uses simple linear algebra computations to verify the duality property.

1.3 Back-annotations

Besides the distinction between code "suppliers" and code "consumers" in the context of ACC, our framework for WCET analysis also makes the obvious distinction between "source" code and "machine" code. The reason is that WCET analysis must be performed on machine code, at hardware level. However, from the perspective of the developer, the WCET estimations can be of great help when provided at source-code level and complemented with a specification mechanism. In this way, the complete development environment of an embedded real-time system would support WCET verification both at source level, on the supplier side, and at machine level, on the consumer side. Fig. 1.3 given an overview of our complete certifying platform for WCET-driven program verification.

The requirement for reasoning on the WCET at source level is the fact that the loss of "abstraction" introduced by the compilation is not definite. The enabling technology that makes this possible in the DWARF *standard* [134]. This standard provides compiler debug information that consists in a bidirectional correspondence between the source-code line numbers and the instruction memory positions which hold the respective machine code. Our support for WCET verification at source level consists in a *back-annotation* mechanism that uses DWARF together with the meta-semantic formalism in order to specify WCET safety requirements in the form of *design contracts* [90].

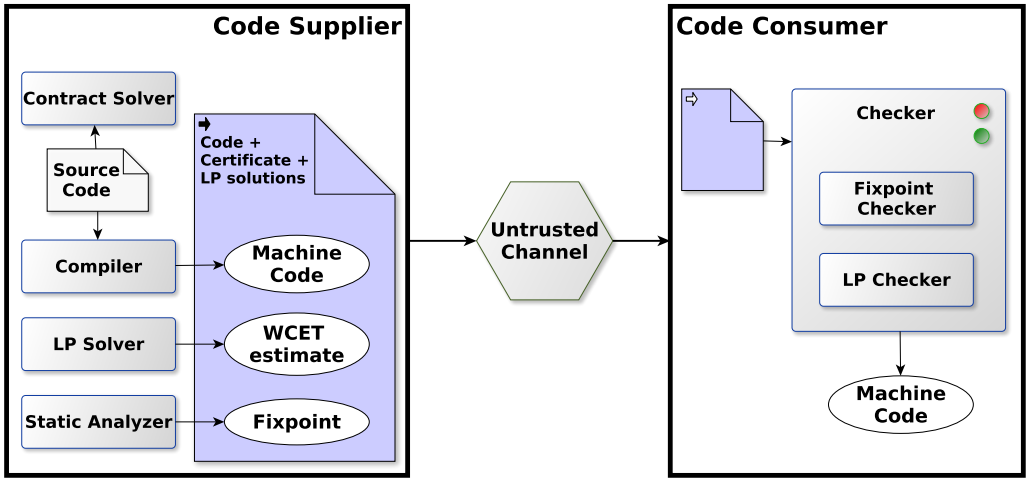


Figure 1.3: Overview of the Certifying Platform

1.4 Contributions

Novel contributions introduced in this thesis are:

i/ The definition of a polymorphic, denotational two-level meta-language [100] capable to express the semantics of different defined languages (e.g. imperative, data-flow, synchronous programming languages) in a unified fixpoint form by means of algebraic relations. The same meta-program can be parametrized by different abstract state transformers, defined at the denotational level for a specific abstract domain.

ii/ The definition of a compositional and generic data-flow analyzer that uses a topological order on the program syntax to instantiate expressions of the meta-language. This is achieved automatically by providing interpretations of an intermediate graph-based language into the λ-calculus. The result is a type-safe fixpoint semantics of expressions using the higher-order combinators of the meta-language for free. Moreover,

the algebraic properties of the meta-language provide transformation rules for the intermediate language.

iii/ An application of the hierarchy of semantics by abstract interpretation proposed in [27] and the notion of weak topological order [20] to abstract the trace program semantics into the formalism of the meta-language. This enables efficient fixpoint computations by means of chaotic iteration strategies and provides a systematic way to apply the *functional approach* [128] to interprocedural analysis.

iv/ For the purpose of WCET analysis, the data-flow analyzer computes simultaneously the *value analysis*, the *cache analysis* and the *pipeline analysis* of machine programs running on an ARM9 processor. *Program flow analysis* is automatically obtained at machine code level as an instrumented *value analysis*.

v/ The abstract interpreters defined at the lower-level of the meta-language are "correct by construction" in the sense that they are obtained by the calculational approach based on Galois connections proposed by Cousot [28]. Correctness proofs are given for the *value analysis* of the abstract instruction semantics and for the *cache analysis* using the *least recently used* (LRU) replacement policy.

vi/ Inclusion of the WCET checking phase inside the ACC framework using the certifying properties of duality theory. The complexity of the optimization problem on the consumer side is reduced from NP-hard (ILP) to polynomial time (LP), by the fact that the verification of optimal solutions can be performed using simple linear algebra computations, without the need to recompute the simplex algorithm.

vii/ The *flow conservation* constraints of the linear programming (LP) problem are obtained as abstract interpretations of the relational semantics of the machine program. Therefore, the preservation of the integer semantics is deducible. Additionally, for the purpose of *path analysis*, the *capacity constraints* of the linear program are automatically given by the *program flow analysis*.

viii/ Integration of a back-annotation mechanism into the ACC framework to perform WCET checking on source code level. Design contracts are specified in the source code using the expressions of the meta-language, which compute state transformations on the information produced at machine code level.

ix/ A computationally feasible extension of the WCET analyzer to multicore systems by means of the $\mathcal{LR}$-server model presented in [131]. This model defines an abstraction of the temporal behavior of application running on different processor cores and provides a compositional WCET analysis. The formalization and implementation of the $\mathcal{LR}$-server model is performed in the context of our data-flow analysis using the higher-order combinators and the abstract interpretation framework based on Galois connections.

x/ To some extent, we demonstrate that Haskell can be used as a language where the mathematical complex notions underlying the theory of abstract interpretation can be easily and elegantly implemented and applied to the analysis of complex hardware architectures.

The previous contributions are fully represented in our four papers [112, 113, 114, 115], which were published during a total period of thesis time of three years. The complete Haskell prototype can be downloaded from `http://www.dcc.fc.up.pt/~vitor.rodrigues/`, under the section Software.

The rest of thesis is organized according to the following structure. We start with three chapter introducing the necessary background material that supports the rest of the thesis. In Chapter 2, we present a summary of the principal topic of domain theory and denotational semantics and in Chapter 3, we give a comprehensive introduction to the theory of abstract interpretation. Finally, we introduce in Chapter 4 the most consensual classification of the several components that can be part of the toolchain of a generic WCET static analyzer.

Our first main contribution is presented in Chapter 5, where we describe a generic data flow framework that can be parametrized to perform static analysis by abstract interpretation. A particular instantiation of this framework for the purpose of WCET analysis is described in detail in Chapter 6. An extension of the ACC framework for performing a semantics-based program verification using the WCET as the safety parameter is presented in Chapter 7. The last contribution on WCET analysis in multicore architectures is described in detail in detail in Chapter 8, where we also give the results of the WCET analysis for a comprehensive subset of the Mälardalen benchmark programs. Finally, we conclude and discuss future work.

Chapter 2

Denotational Semantics

The present chapter introduces the necessary background on domain theory and denotational semantics. Basic notions of domain theory, such as monotonicity, least upper bound, continuity and least fixpoints, are used to define the denotational semantics of a program with looping and recursive constructs. In particular, we describe the Kleene's constructive approach to fixpoint semantics that constitutes the foundations of our generic data flow analysis framework presented in Chapter 5. The notions of complete partial orders and lattices are also a prerequisite for understanding the theory of abstract interpretation, in particular the notion of Galois connection introduced in Chapter 3.

The design of programming languages is generally organized by a syntax and a semantics. The *semantics* gives the meaning of syntactically correct programs. Here, when we refer to syntactic entities, we refer to their *abstract syntax* as the means to specify parse trees in the language. The construction of the unique parse trees is guided by the *concrete syntax* of the programming language. On the other hand, the semantics of the programming language is specified by defining *semantic functions* for each of syntactic entities, so that the meaning, or denotation, of a syntactic entity is obtained by passing it as argument to the semantic function which then returns its meaning, that is, the *effect* of the function's evaluation. Denotations are well-defined mathematical objects and can often be higher-order functions.

In fact, the meaning of programs in the denotational setting is modeled by mathematical functions that represent the *effect* of interpreting the syntactical constructs [101]. Such effect is commonly expressed in typed λ-calculus. According to Stoy [132], the *denotational assumption* is that the meaning of a possible composite syntactic elements is a mathematical combination of the sub-elements in the abstract syntax-tree. Hence, a denotational definition consists in the following parts: (1) a collection of domain definitions denoted by the types in the λ-expressions; (2) a collection of semantic functions, usually one for each syntactic category; and (3) a collection of semantic rules that express the meanings of syntactic composite elements in terms of the meanings of their substructures, usually by λ-expressions.

The principle of compositionality in denotational semantics is a powerful mechanism that enables the generalization of the denotational framework to allow the definition of effects using non-standard interpretations in addition to the standard semantics. In this way, a denotational language definition can be factorized in two parts: (1) a *core semantics* containing semantic rules and their types, where some function symbols and domain names remain uninterpreted; and (2) an *interpretation* giving meaning to the missing definitions of domain names and function symbols.

In the same line of thought, the general applicability of the framework of denotational semantics can enhanced by a denotational meta-language [70]. In this way, instead of regarding denotational semantics as the direct mapping of syntactic categories to mathematical domains, the denotational semantics is factored through the meta-language. In the first phase, the syntactic categories are denoted by the terms of the meta-language. Secondly, these terms are interpreted as elements in the mathematical domains used in denotational semantics.

The next sections describe the essential background on denotational semantics. First we give the foundations for domain theory and next we describe the compositional computational model of fixpoints used in denotational semantics [42, 85]. Finally, we briefly describe the two-level meta-language proposed by Nielson in [99, 100].

2.1 Domain Theory

A *partiallly order set*, commonly designated by *poset*, is denoted by $S(\sqsubseteq)$, where S is a set and $\sqsubseteq$ is a binary relation on S that is *reflexive* ($\forall x \in S : x \sqsubseteq x$), *transitive* ($\forall x, y, z \in S : x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$) and *antisymmetric* ($\forall x, y \in S : x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$). Informally, $x \sqsubseteq y$ means that x *shares its information with* y. For example, if the binary relation is the *subset relation* $\subseteq$, we say that the subset $\{1, 2\}$ *shares* information with the subset $\{1, 2, 3\}$.

Let $S(\sqsubseteq)$ be a poset with partial order $\sqsubseteq$ on a set S. u is an *upper bound* of a subset X of S if and only if u is greater than or equal to all members of X ($\forall x \in X : x \sqsubseteq u$). We say that the element u *summarizes all the information* of X. For example, let 1,2 and 3 be the elements of the set S. When considering upper bounds of subsets X with two elements of S, there are three possible *upper* bounds: $\{1, 2\}$, $\{1, 3\}$ and $\{2, 3\}$.

The *least upper bound* u of a subset X of S is an upper bound of X that is smaller than any other upper bound of X ($\forall x \in X : x \sqsubseteq u \wedge u' \in S : (\forall x \in X : x \sqsubseteq u') \Rightarrow (u \sqsubseteq u')$). Therefore, the least upper bound of X adds as little extra information as possible to that already present in the elements of X. For any $x, y \in S$, we write $x \sqcup y$ to define the least upper bound on the set $\{x, y\}$ such that $x \sqsubseteq y \Leftrightarrow x \sqcup y = y$. A least upper bound is unique. If it exists, the least upper bound of $X \subseteq S$ is written $\bigsqcup X$. For the same example above,

the least upper bound of $S(\subseteq)$, i.e. the subset that summarizes *all* the information of S, understood as set inclusion, is unique and equal to $\bigcup S = \{1, 2, 3\}$.

The *lower bounds* and the *greatest lower bound* $\bigsqcap X$ of a subset $X \subseteq S$ are *dual* (i.e. their definition is obtained from that of upper bounds and least upper bound by replacing $\sqsubseteq$ by its dual $\sqsupseteq$). For any $x, y \in S$, we write $x \sqcap y$ to define the greatest lower bound on the set $\{x, y\}$ such that if $x \sqsubseteq y$, then $x \sqcap y = x$. The smallest element of a set S is called the *infimum* and is defined by $\bot = \bigsqcap S$. Intuitively, the infimum contains *no information*. On the other hand, the greatest element of S is called the *supremum* and is defined by $\top = \bigsqcup S$.

Let $P(\sqsubseteq)$ be a poset with partial order $\sqsubseteq$ on a set P. A *complete lattice* $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ is a poset $L(\sqsubseteq)$ such that any subset X of L has a least upper bound $\bigsqcup X$ and a greatest lower bound $\bigsqcap X$. In particular, the smallest element is $\bot$ whilst the greatest is $\top$.

A subset $X \subseteq P$ is called a *chain* if, for any two elements of X, one shares information with the other, i.e. any two elements of P are comparable: $\forall x, y \in P : x \sqsubseteq y \vee y \sqsubseteq x$. A poset $P(\sqsubseteq)$ satisfies the *ascending chain condition* if given any sequence $x_1 \sqsubseteq x_2 \sqsubseteq \ldots x_n \sqsubseteq \ldots$ of elements of P, there exists $k \in \mathbb{N}$ such that $x_k = x_{k+1} = \ldots$. An poset P is a *cpo* (abbreviation for *complete partial order*) if every increasing chain has a least upper bound $\bigsqcup X$. More specifically, a *strict* cpo has an *infimum*, typically $\bot$, denoting absence of information.

Let S and T be given sets. The *powerset* $\wp(S)$ is the set $\{X \mid X \in S\}$ of all subsets of S. The *cartesian product* $S \times T$ is the set $\{\langle s, t\rangle \mid s \in S \wedge t \in T\}$. A binary relation on $S \times T$ is a subset $\rho \in \wp(S \times T)$ of $S \times T$, that is, $\rho \subseteq (S \times T)$.

We write $\varphi \in S \hookrightarrow T$ to mean that φ is a *partial function* from S to T, i.e. a relation $\varphi \in \wp(S \times T)$ such that the binary relation $\langle s, t\rangle \in \varphi$ only if $s \in S$ and $t \in T$ and, for every $s \in S$, there exists at most one $t \in T$, written $\varphi[\![s]\!]$, $\varphi[s]$, or $\varphi(s)$, satisfying $\langle s, t\rangle \in \varphi$. We write $\varphi \in S \mapsto T$ to mean that φ is a *total function* of S into T i.e. $\varphi(s)$ exists for all $s \in S$ $(\forall s \in S : \exists t \in T : \langle s, t\rangle \in \varphi)$. As usual, functional composition $\circ$ is defined by $\varphi \circ \psi(s) = \varphi(\psi(s))$. The *image* of $X \subseteq S$ by $\varphi \in S \mapsto T$ is $\varphi^*(X) = \{\varphi(x) \mid x \in X\}$.

Let $P(\sqsubseteq, \bigsqcup)$ be a poset with least upper bound $\bigsqcup$ and $Q(\preceq, \bigvee)$ be a poset with least upper bound $\bigvee$. $P(\sqsubseteq) \overset{m}{\longmapsto} Q(\preceq)$ denotes the set of total functions $\varphi \in P \mapsto Q$ that are *monotone*, i.e. order morphisms: $\forall x \in P : \forall y \in Q : x \sqsubseteq y \Rightarrow \varphi(x) \preceq \varphi(y)$. The intuition is that if x shares its information with y, then when φ is applied to x and y, a similar relationship holds between the images of the functions.

Let $P(\sqsubseteq, \bigsqcup)$ be a poset with least upper bound $\bigsqcup$ and $Q(\preceq, \bigvee)$ be a poset with least upper bound $\bigvee$ and let $\varphi \in P \overset{m}{\longmapsto} Q$ be a monotone function. If $X \subseteq P$ is a chain, then $\varphi^*(X)$ is a chain in Q. Furthermore, given the ascending chain $x_1 \sqsubseteq x_2 \sqsubseteq \ldots x_n \sqsubseteq \ldots$, we have that φ is monotone (or alternatively, order-preserving) iff $\bigvee_{n \geqslant 0} \varphi(x_n) \preceq \varphi(\bigsqcup_{n \geqslant 0})$. However, in general, a monotone function does not preserve least upper bounds on chains.

Let $P(\sqsubseteq, \bigsqcup)$ be a poset with least upper bound $\bigsqcup$ and $Q(\preceq, \bigvee)$ be a poset with least upper bound $\bigvee$. $P(\sqsubseteq, \bigsqcup) \overset{c}{\longmapsto} Q(\bigvee)$ denote the set of total functions $\varphi \in P \mapsto Q$ that are *upper-continuous* i.e., which preserve existing least upper bounds of increasing chains. Since the least upper bound is necessarily unique, we have that if $X \subseteq Q$ is an increasing chain for $\sqsubseteq$ and $\bigsqcup X$ exists then $\varphi(\bigsqcup X) = \bigvee \varphi^*(X)$.

Let $P(\sqsubseteq, \bigsqcup)$ be a poset with least upper bound $\bigsqcup$ and $Q(\preceq, \bigvee)$ be a poset with least upper bound $\bigvee$. $P(\bigsqcup) \overset{a}{\longmapsto} Q(\bigvee)$ denote the set of total functions $\varphi \in P \mapsto Q$ that are *additive* i.e., complete join morphisms preserving least upper bounds of arbitrary subsets, when they exist: if $X \subseteq P$ and $\bigsqcup X$ exists then $\varphi(\bigsqcup X) = \bigvee \varphi^*(X)$. When the above notions are restricted to sets equipotent with the set $\mathbb{N}$ of natural numbers, they are qualified by the attribute w as in *w-chain*, *w-cpo*, *w-continuity*, etc.

A *functional* $F \in [P(\sqsubseteq) \overset{m}{\mapsto} P(\sqsubseteq)]$ maps the set of functions $[P(\sqsubseteq) \overset{m}{\mapsto} P(\sqsubseteq)]$ into itself, i.e F takes as argument any monotonic function $\varphi \in P \mapsto P$ and yields a monotonic function $F(f)$ as its value. A functional F is said to be monotonic if $\varphi \sqsubseteq \phi$ implies $F(\varphi) \sqsubseteq F(\phi)$ for all $\varphi, \phi \in [P(\sqsubseteq) \overset{m}{\mapsto} P(\sqsubseteq)]$. Let $P(\sqsubseteq, \sqcup)$ be a poset with a least upper bound $\sqcup$ and φ^n be a chain of functions $\varphi_0 \dot{\sqsubseteq} \varphi_1 \dot{\sqsubseteq} \varphi_2 \dot{\sqsubseteq} \varphi_3 \ldots$, where $\dot{\sqsubseteq}$ is the point-wise ordering on φ. Then, a monotonic functional F is said to be continuous if we have that $F(\sqcup\varphi^n) = \sqcup F(\varphi)^n$.

2.2 Fixpoints

A *fixpoint* $x \in P$ of $\varphi \in P \mapsto P$ is such $\varphi(x) = x$. We write $\mathit{fix}\,\varphi$ for the set $\{x \in P \mid \varphi(x) = x\}$ of fixpoints of φ. The *least fixpoint* $\mathit{lfp}\,\varphi$ of φ is the unique $x \in \mathit{fix}\,\varphi$ such that $\forall y \in \mathit{fix}\,\varphi : x \sqsubseteq y$. The dual notion is that of *greatest fixpoint* $\mathit{gfp}\,\varphi$. The n-fold composite, φ^n, of a map $\varphi : P \to P$ is defined as follows: φ^n is the identity map if $n = 0$ and $\varphi^n = \varphi \circ \varphi^{n-1}$ for $n \geqslant 1$. If φ is monotone so is φ^n.

Let $P(\sqsubseteq)$ be a cpo and let $\varphi \in P \mapsto P$ be a continuous map. Applying the monotone map φ^n, we have that $\varphi^n(\bot) \sqsubseteq \varphi^{n+1}(\bot)$, for all n and the following increasing chain:

$$\bot \sqsubseteq \varphi(\bot) \sqsubseteq \cdots \sqsubseteq \varphi^n(\bot) \sqsubseteq \varphi^{n+1}(\bot) \sqsubseteq \ldots$$

Then, according to the Kleene first-recursion theorem [74], the least fixpoint $\mathit{lfp}\,\varphi$ exists and equals to $\bigsqcup_{n \geqslant 0} \varphi^n(\bot)$.

By Tarski's fixpoint theorem [133], the fixpoints of a monotone mapping $\varphi \in L(\sqsubseteq) \overset{m}{\longmapsto} L(\sqsubseteq)$ on a complete lattice $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ form a complete lattice $\mathit{fix}\,\varphi$ for $\sqsubseteq$ with infimum $\mathit{lfp}\,\varphi = \bigsqcap \varphi^{\sqsubseteq}$ and supremum $\mathit{gfp}\,\varphi = \bigsqcup \varphi^{\sqsupseteq}$ where $\varphi^{\sqsubseteq} = \{x \in L \mid \varphi(x) \sqsubseteq x\}$ is the set of *post-fixpoints* and $\varphi^{\sqsupseteq} = \{x \in L \mid \varphi(x) \sqsupseteq x\}$ is the set of *pre-fixpoints* of φ. Fig. 2.1(a) illustrates the complete lattice $\mathit{fix}\,\varphi$ for $\sqsubseteq$.

Let F be a *functional* over $[P(\sqsubseteq) \overset{m}{\mapsto} P(\sqsubseteq)]$ [85]. We say that a function $\varphi \in [P(\sqsubseteq) \overset{m}{\mapsto} P(\sqsubseteq)]$

is a fixpoint of F if $F(\varphi) = \varphi$, i.e. if F maps the function φ into itself. If φ is a fixpoint of F and $\varphi \sqsubseteq \phi$ for any other fixpoint ϕ of τ, then φ is called the least fixpoint of τ.

Let F be any monotonic functional over $[P(\sqsubseteq) \overset{m}{\mapsto} P(\sqsubseteq)]$ and $\Omega \sqsubseteq F(\Omega) \sqsubseteq F^2(\Omega) \sqsubseteq \ldots$ be an increasing chain, where $F^0(\Omega)$ stands for Ω (the totally undefined function) and $F^{i+1}(\Omega)$ is $F(F^i(\Omega))$ for $i \geqslant 0$. Then, the $F^i(\Omega)$ has a least upper bound. By Kleene's first recursion theorem, every continuous functional F has a least fixpoint denoted by φ_F defined as $\varphi_F = \bigsqcup_{i \geqslant 0} F^i(\Omega)$. In fact, since F is continuous:

$$F(\varphi_F) = F(\bigsqcup_{i \geqslant 0} F^i(\Omega)) = \bigsqcup_{i \geqslant 0} F^{i+1}(\Omega) = \bigsqcup_{i \geqslant 0} F^i(\Omega) = \varphi_F \tag{2.1}$$

The Tarski's theorem states that in any complete lattice L and for any monotone function $f : L \to L$ there exists a least fixpoint that coincides with the smallest post-fixpoint. However, this theorem does not particularize any method to compute such fixpoint. Alternatively, if f is continuous, the Kleene theorem provides a constructive approach to fixpoint computation, where the least fixpoint is the least upper bound of the ascending Kleene chain of Def. (2.1), as Fig. 2.1(b) shows. In this case, infinite strictly increasing chains are forbidden. Although constructive versions of Tarski's theorem exist [133], we prefer the Kleene approach as it provides the foundations for the definition of a *fixpoint operator*, commonly termed *FIX*.

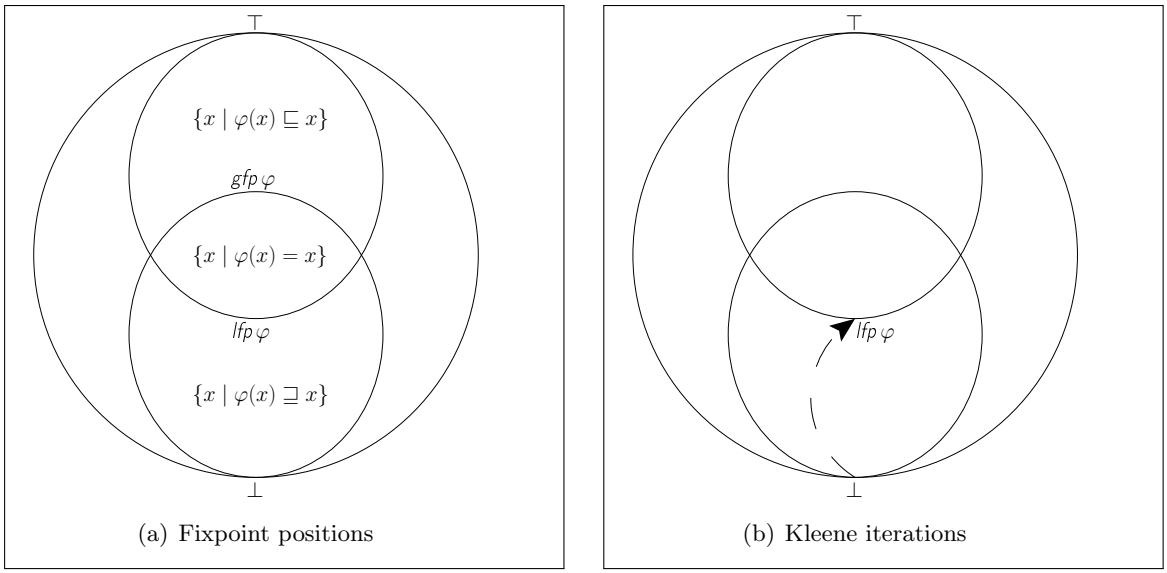


Figure 2.1: Fixpoint positions in a complete lattice

Let $\varphi \in P(\sqsubseteq) \overset{c}{\longmapsto} P(\sqsubseteq)$ be a continuous function on the strict cpo P with least element $\bot$. The Kleene's least fixpoint $lfp\,\varphi$ follows from the fact that by finitely applying n times φ^n we obtain $\varphi^n(\bot) \sqsubseteq \varphi^n(lfp\,\varphi) = lfp\,\varphi$. This can be computationally achieved by applying the fixpoint operator *FIX* to φ:

$$FIX(\varphi) = \bigsqcup\{\varphi^n \bot \mid n \geqslant 0\} \tag{2.2}$$

When $n = 0$ we have $\varphi^0(\bot) = \bot$. Additionally, by induction on n, we have that $\varphi^{n+1} = \varphi \circ \varphi^n$. By definition of strict cpo, we have $\bot \sqsubseteq p$ for all $p \in P$, whence by monotonicity of φ, $\varphi^n(\bot) \sqsubseteq \varphi^n(p)$. It follows that $\varphi^n(\bot) \sqsubseteq \varphi^m(\bot)$, whenever $n \leqslant m$. From this, $\{\varphi^n(\bot) \mid n \geqslant 0\}$ is a non-empty chain in P and $FIX(\varphi)$ exists because P is a cpo.

To prove that the function $FIX(\varphi)$ computes the least fixpoint $lfp\,\varphi$, we first need to show that $FIX(\varphi)$ is a *fixpoint*. i.e. that $\varphi(FIX(\varphi)) = FIX(\varphi)$. We calculate:

$$\begin{aligned}
\varphi(FIX(\varphi)) &= \varphi(\bigsqcup_{n\geqslant 0} \varphi^n(\bot)) && \text{(definition of } FIX(\varphi)) \\
&= \bigsqcup_{n\geqslant 0} \varphi(\varphi(\bot)) && \text{(continuity of } \varphi) \\
&= \bigsqcup_{n\geqslant 1} \varphi(\bot) \\
&= \bigsqcup(\{\varphi(\bot) \mid n \geqslant 1\} \cup \{\bot\}) && (\bigsqcup(Y \cup \{\bot\}) = \bigsqcup Y \\
& && \text{for all chains } Y) \\
&= \bigsqcup_{n\geqslant 0} \varphi^n(\bot) && (\varphi^0(\bot) = \bot) \\
&= FIX(f)
\end{aligned}$$

Additionally, $FIX(\varphi)$ is the *least* fixed point. Assume that $p \in P$ belongs to the set of *post-fixpoints*. Since P is a strict cpo, we have $\bot \sqsubseteq p$, whence the monotonicity of φ gives $\varphi^n(\bot) \sqsubseteq \varphi^n(p)$. If p is a fixpoint, we obtain $\varphi^n(\bot) \sqsubseteq p$ for all $n \geqslant 0$. Hence p is an upper bound of the chain $\{\varphi^n(\bot) \mid n \geqslant 0\}$, and if by hypothesis $FIX(\varphi)$ is the least upper bound, then $FIX(\varphi) \sqsubseteq p$.

Note that in the definition of FIX (2.2), the argument φ can be a functional. Assume that φ is a recursive function with an exit condition. In order to compute the fixpoint of φ one needs to define an anonymous function f that constitutes an argument of φ at the same time it appears in the body of φ. The use of FIX allows us to unravel the definition of $\varphi(f)$: every time f recurs, another iteration of φ is obtained via $FIX(\varphi) = \varphi(FIX(\varphi))$, which behaves as the next call in the recursion chain. After unravelling the definition of $\varphi(f)$, the exit condition will eventually occur, i.e. a program clause where f does not take place, which then initiates the evaluation of the recursive chain until the least fixpoint is found.

The Haskell definition of FIX is given by the function *fix*. The type variable a is polymorphic and can represent a higher-order type, as required when the argument g is a functional [101]. Note that the FIX operator is based on the notion of *general recursion*, in which non-termination is semantically valid. Nonetheless, if the least fixpoint of g does exist, the return type a clearly specifies that the Kleene's recursive chain has a least upper bound.

```
fix :: (a → a) → a
fix g = g (fix g)
```

2.3 Two-Level Denotational Meta-Language

As already mentioned, the goal of denotational semantics [54, 132] is not to specify *how* a program is executed, but merely to specify the *effect* of executing a program. In this context, the distinction between compile-time and run-time made on the two-level denotational meta-language (TML) presented in [100] introduces an important level of modularity to denotational definitions. In particular, this distinction is important for the efficient implementation of programming languages with the objective to automatically generate compilers.

The use of two-level denotational semantics in data flow analysis by abstract interpretation is originally found in the work of Nielson [70, 95, 96]. The two-level meta-language paved the way for systematic treatment of data flow analysis, where the correctness relation between different abstract interpretations is obtained through the comparison of different run-time denotational definitions, all sharing the same compile-time denotational definition. Similarly, code generation for various abstract machines can be specified by providing different run-time interpretations of the meta-language [99].

The traditional denotational approaches of Gordon and Stoy [54, 132], use a typed λ-calculus as meta-language with the following abstract syntax for types:

$$\mathrm{t} ::= \mathrm{A} \mid \mathrm{t}_1 \times \cdots \times \mathrm{t}_k \mid \mathrm{t}_1 + \cdots + \mathrm{t}_k \mid \mathrm{rec\,X.t} \mid \mathrm{X} \mid \mathrm{t}_1 \to \mathrm{t}_2$$

The two-level meta-language is obtained by distinguishing between compile-time types (ct) and run-time types (rt). Hence, the meta-language TML has the following types:

$$\mathrm{ct} ::= \mathrm{A} \mid \mathrm{ct}_1 \times \cdots \times \mathrm{ct}_k \mid \mathrm{ct}_1 + \cdots + \mathrm{ct}_k \mid \mathrm{rec\,X.ct} \mid \mathrm{X} \mid \mathrm{ct}_1 \to \mathrm{ct}_2 \mid \mathrm{rt}_1 \underline{\to} \mathrm{rt}_2$$

$$\mathrm{rt} ::= \underline{\mathrm{A}} \mid \mathrm{ct}_1 \mathbin{\underline{\times}} \ldots \mathbin{\underline{\times}} \mathrm{ct}_k \mid \mathrm{ct}_1 \mathbin{\underline{+}} \ldots \mathbin{\underline{+}} \mathrm{ct}_k \mid \underline{\mathrm{rec\,X}}.\mathrm{rt} \mid \underline{\mathrm{X}}$$

The underlining is used to disambiguate the syntax of the two levels. The purpose of the meta-language is strongly focused on the run-time functional types, $\mathrm{rt}_1 \underline{\to} \mathrm{rt}_2$, because they synthesize aspects such as "semantic transformations" and "piece of code" which are applicable in abstract interpretation as well as code generation. Intuitively, the "static" expressions in compile-time types can be parametrized with different run-time entities. However, interaction between the two type levels is restricted by the absence of $\mathrm{rt} ::= \mathrm{ct}$ meaning that, as expected, compile time entities cannot be discussed at run-time. On the other hand, the run-time type $\mathrm{rt}_1 \underline{\to} \mathrm{rt}_2$ can be discussed at compile-time (ct).

Expressions of the two-level meta-language are defined by:

$$\begin{aligned}
\mathrm{e} ::= {} & \mathrm{f} \mid (\mathrm{e}_1, \ldots, \mathrm{e}_k) \mid \mathrm{e} \downarrow \mathrm{j} \mid \mathrm{in_j\,e} \mid \mathrm{is_j\,e} \mid \mathrm{out_j\,e} \\
& \mid \lambda(\mathrm{x{:}ct}).\mathrm{e} \mid \mathrm{e}_1(\mathrm{e}_2) \mid \mathrm{x} \mid \mathrm{mkrec\ e} \mid \mathrm{unrec\ e} \\
& \mid \mathrm{e} \to \mathrm{e}_1, \mathrm{e}_2 \mid \mathrm{fix_{ct}\,e} \\
& \mid \underline{\mathrm{tuple}}\,(\mathrm{e}_1, \ldots, \mathrm{e}_k) \mid \underline{\mathrm{take_j}} \mid \underline{\mathrm{in_j}} \mid \underline{\mathrm{case}}\,(\mathrm{e}_1, \ldots, \mathrm{e}_k) \\
& \mid \underline{\mathrm{mkrec}} \mid \underline{\mathrm{unrec}} \mid \underline{\mathrm{cond}}\,(\mathrm{e}, \mathrm{e}_1, \mathrm{e}_2) \mid \mathrm{e}_1 \circ \mathrm{e}_2
\end{aligned}$$

A constant of type ct is denoted by f, but not all compile-time types are allowed. There must not be run-time function spaces in the domain of a compile-time function space in ct and, in addition, ct must not contain free type variables. Therefore, (λ(x:ct).e) is a compile-time expression denoting a lambda abstraction, where x is a compile-time type variable (ct) occurring bound in the expression e.

The semantics of the meta-language is given by an *interpretation* $\underline{\mathrm{I}}$ consisting of two parts, a type part and an expression part. The interpretation $\underline{\mathrm{I}}[\![\mathrm{ct}]\!]$ is a cpo that defines the type part for each closed type ct [95]. When the interpretation is the *standard* interpretation, the cpo's are obtained by interpreting $\times$ as cartesian products, $\underline{\times}$ as smash product, $+$ as separate sum, $\underline{+}$ as coaleased product, $\rightarrow$ as function space and $\underline{\rightarrow}$ as strict function space. The recursive domain equations $\mathrm{rec\,X.ct}$ and $\underline{\mathrm{rec\,X}}$.rt are solved using the categorical approach given in [129]. Interpretations of expressions of the meta-language are defined by the function $\underline{\mathrm{I}}[\![\mathrm{e}]\!] : \underline{\mathrm{I}}[\![\mathrm{ct}_1]\!] \times \cdots \times \underline{\mathrm{I}}[\![\mathrm{ct}_1]\!] \rightarrow \underline{\mathrm{I}}[\![\mathrm{ct}_1]\!]$ for each well-typed expression e.

Example 1. Examples of standard interpretations of TML terms.
Next, we exemplify the standard interpretation of the conditional expression $\underline{\mathrm{cond}}$ and the higher-order functional composition ($\circ$). These interpretations are particularly relevant to our approach since they express generic control-flow combinators. Assuming that the base types $\underline{\mathrm{A}}$ include the truth values $\underline{\mathrm{T}}$, the combinator style expressions $\underline{\mathrm{cond}}$ and ($\circ$), are interpreted in the standard interpretation $\mathcal{S}$ as follows:

$$\underline{\mathcal{S}}(\underline{\mathrm{cond}}) : (\underline{\mathcal{S}}[\![\mathrm{rt}_1 \underline{\rightarrow} \underline{\mathrm{T}}]\!] \times \underline{\mathcal{S}}[\![\mathrm{rt}_1 \underline{\rightarrow} \mathrm{rt}_2]\!] \times \underline{\mathcal{S}}[\![\mathrm{rt}_1 \underline{\rightarrow} \mathrm{rt}_2]\!]) \rightarrow \underline{\mathcal{S}}[\![\mathrm{rt}_1 \underline{\rightarrow} \mathrm{rt}_2]\!]$$
$$\underline{\mathcal{S}}(\underline{\mathrm{cond}}) = \lambda(\mathrm{E}, \mathrm{E}_1, \mathrm{E}_2).\lambda v.(\mathrm{E}\, v \rightarrow \mathrm{E}_1\, v, \mathrm{E}_2\, v)$$
$$\underline{\mathcal{S}}(\circ) : \underline{\mathcal{S}}[\![\mathrm{rt}_1 \underline{\rightarrow} \mathrm{rt}_2]\!] \times \underline{\mathcal{S}}[\![\mathrm{rt}_1 \underline{\rightarrow} \mathrm{rt}_2]\!] \rightarrow \underline{\mathcal{S}}[\![\mathrm{rt}_1 \underline{\rightarrow} \mathrm{rt}_2]\!]$$
$$\underline{\mathcal{S}}(\circ) = \lambda(\mathrm{E}_1, \mathrm{E}_2).\lambda v.(\mathrm{E}_1(\mathrm{E}_2\, v))$$

▲

As already mentioned, at the compile-time level of the meta-language we cannot reason about run-time values, but only about transformations on run-time values, each of which of type $\mathrm{rt}_1 \underline{\rightarrow} \mathrm{rt}_2$. The intuition of the two-level meta-language is that such transformations can also be obtained by executing a piece of code on an appropriate abstract machine. Therefore, code generation can be achieved by specifying new interpretations for the basic expressions of type $\mathrm{rt}_1 \underline{\rightarrow} \mathrm{rt}_2$.

Chapter 3

Abstract Interpretation

The present chapter introduces the necessary background on the theory of abstract interpretation. Key concepts such as abstract fixpoint semantics, Galois connections, higher-order Galois connections and fixpoint induction using Galois connections are described in detail. A brief introduction is made to the *merge over all paths* (MOP) and *minimum fixed point* (MFP) fixpoint solutions and to the algorithms that are able to efficiently compute them, namely *chaotic fixpoint algorithms* using particular *iteration strategies*. These concepts are applied in the definition of our generic data flow framework presented in Chapter 5.

A Galois connections is used to establish a correspondence between two posets typically referred to as the *concrete* and the *abstract* domains. Let $P(\sqsubseteq)$ and $Q(\preceq)$ be two posets. A Galois connection between P and Q consists of two monotone functions, $\alpha : P \mapsto Q$ and $\gamma : Q \mapsto P$ such that for all $p \in P$ and $q \in Q$, we have $\alpha(p) \preceq q$ if and only if $p \sqsubseteq \gamma(q)$. Higher-order Galois connections are used to induce abstract semantic transformers, taking as starting point a formal specification of the concrete semantics of a program. Several abstract domains and the corresponding calculational approaches to the induction of abstract interpreters are described in Chapter 6.

The denotational semantics in the style of Scott and Strachey [132] provides a "mathematical semantics" for the design of programming languages where programs *denote* computations in some universe of objects. Abstract interpretation [27, 31, 35, 37, 32] is a general framework allowing the certification of dynamic properties of programs and can be proved to be consistent with the formal semantics (not necessarily denotational) of the language. The hallmark of abstract interpretation is the possibility to use such denotations [1, 96, 120], to describe computations in another universe of abstract objects, with the objective to obtain some information on the actual computations, albeit statically and at compile time.

The essential idea is the formulation of a correspondence between *abstract* values and *concrete* values. The concern of abstract interpretation is with a particular structure of an abstract universe of computations where program *properties* are defined. These properties

give a summary of some facets of the actual executions of a program, but it is, in general, an inaccurate process. Nonetheless, the consequent incompleteness of an abstract interpretation is tolerable when the answer the programmer expects does not need to be fully exact or when the formal specification of program properties contemplate *false alarms* [29, 39], i.e. when the verification of the specification gives a negative answer in presence of false positives.

One possible way to perform abstract interpretations consists in the use of a transitive closure mechanism, defined for the basic operations of the language [31]. The premise that the abstract interpretation can be fully worked out at compile time implies the existence of fixpoint iterative methods able to compute program properties after finitely many steps. However, it is not a requirement to have the abstract values belonging to finite Kleene sequences [32]. In such cases, the *widening* and *narrowing* operators [38] can be used to accelerate the convergence of the fixpoint computation performed by the transitive closure mechanism.

Data flow analysis can be formalized as abstract interpretation of programs by letting the basic operations of the programming language be locally interpreted by order preserving, i.e monotone, functions. In general, program properties are modelled in complete semilattices [42]. In this way, and given a formal model of the program syntax, a program is associated to a system of recursive semantic equations. Given that the theory of abstract interpretation provides local consistency, the global program properties are defined as one of the post-fixpoints of that system [133].

3.1 Abstract Values

Abstract interpretation performs a "symbolic" interpretation of a program using abstract values instead of concrete or execution values. The definition of abstract value is an extensional definition in the sense that it denotes a set of concrete values or properties of such a set in a consistent way. In its turn, the definition of concrete value is an intensional definition since it corresponds to the *collecting semantics* of the program at a particular program point. The role of the collection semantics is to provide a sound and relatively complete proof method for the considered class of properties. Whence, it is typically defined as the powerset lattice 2^D of the concrete domain D. The consistency between concrete and abstract values may be rigorously defined by an abstraction function α and, inversely, by a concretization function γ.

Let $D(\subseteq, \bigcup)$ be the powerset of concrete values and $D^\sharp(\sqsubseteq, \bigsqcup)$ a complete join-semilattice of abstract values under the partial ordering $\sqsubseteq$. The functions α and γ are order-preserving, i.e. monotone, functions defined as:

$$\alpha \in 2^D \mapsto D^\sharp \tag{3.1}$$

$$\gamma \in D^\sharp \mapsto 2^D \tag{3.2}$$

In the particular case where the abstraction function α preserves existing least upper bounds, i.e. α is a complete join morphism, the concretization function γ is uniquely defined [37]. Hence, the pair $\langle \alpha, \gamma \rangle$ is a Galois connection meaning that:

$$\forall p^\sharp \in D^\sharp : \forall P \in 2^D : \alpha(P) \sqsubseteq p^\sharp \Leftrightarrow P \subseteq \gamma(p^\sharp) \tag{3.3}$$

where the following properties are satisfied:

$$\forall p^\sharp \in D^\sharp : \alpha(\gamma(p^\sharp)) \sqsubseteq p^\sharp \tag{3.4}$$

$$\forall P \in D : P \subseteq \gamma(\alpha(P)) \tag{3.5}$$

Def. (3.3) derives from the *extension* of the function α and the *intension* of the set P by stating that an abstract value $p^\sharp$ is the most precise over-approximation of the concrete value P, but not only of P. Def. (3.4) states that the concretization function γ introduces no loss of information. Def. (3.5) introduces the concept of approximation in the sense that the abstraction function $\alpha(P)$ may introduce some loss of information so that when concretizing again $\gamma(\alpha(P))$, a larger set can be obtained.

The abstraction function α is a complete join morphism from $(2^D, \bigcup)$ into $(D^\sharp, \bigsqcup)$ such that $\forall (x, y) \subseteq D^2$, we have $\alpha(x \cup y) = \alpha(x) \sqcup^\sharp \alpha(y)$. This implies that $\sqcup^\sharp$ has associativity, commutativity and idempotency properties, and that the zero element $\bot^\sharp$ of $\bigsqcup$ is also $\alpha(\emptyset)$, where $\emptyset$ is the empty set in 2^D.

3.2 Abstract Semantics

Abstract environments are used to hold the bindings of program variables to their abstract values at every program point. Let $\mathcal{V}$ denote the set of program variables. In general, an abstract environment $\rho \in \mathcal{E}$ is a partial function from variables $v \in \mathcal{V}$ to program properties $p^\sharp \in D^\sharp$. A partial map ρ have the type $\mathcal{V} \hookrightarrow D^\sharp$. Instances of abstract environments consist of a set of tuples $(v, p^\sharp)$, where $\rho(v) = p^\sharp$. The local consistency of the abstract interpretation results from the fact that in every actual execution of the program, the concrete values accessed by v will be in the set $\gamma(p^\sharp)$ at every program point. The set $\mathcal{E}$ of all possible abstract environments is defined as:

$$\mathcal{E} \in 2^{\mathcal{V} \hookrightarrow D^\sharp} \tag{3.6}$$

The point-wise join $\rho_1 \dot{\sqcup}^\sharp \rho_2$ of two abstract environments is defined by:

$$\rho_1 \dot{\sqcup}^\sharp \rho_2 = \{(v, p^\sharp) \mid v \in \mathcal{V} \wedge p^\sharp \in D^\sharp \wedge p^\sharp = \rho_1(v) \sqcup^\sharp \rho_2(v)\} \tag{3.7}$$

The state set Σ consists on the set of all information configurations that can occur during an abstract interpretation. The type of program states is a partial function from a node label $n \in \mathcal{N}$, uniquely identifying a program point, to an abstract environment $\rho \in \mathcal{E}$. For sake of convenience, a configuration $\sigma \in \Sigma$ is instantiated as a set of tuples (n, ρ), where $\rho = \sigma(n)$:

$$\Sigma \in 2^{\mathcal{N} \hookrightarrow \mathcal{E}} \tag{3.8}$$

where the state pairs differ from one another in their labels:

$$\begin{aligned} &\forall \sigma \in \Sigma, \quad \forall (i,j) \in \mathcal{N}^2, \quad \forall (v,u) \in \mathcal{E}^2, \\ &\{(i,v) \in \sigma \wedge (j,u) \in \sigma \wedge (i,v) \neq (j,u)\} \implies i \neq j \end{aligned} \tag{3.9}$$

The state transition function is a total function defining for each state vector the *next* state vector:

$$next \in \Sigma \mapsto \Sigma \tag{3.10}$$

For all program labels $\mathcal{N}$ contained in some state vector $\sigma \in \Sigma$, the initial value of the associated abstract environment $\rho \in \mathcal{E}$ is $\bot^\sharp$. Let $\bot_\sigma$ denote the initial state vector. A "computation sequence" with initial state $\bot_\sigma$ is the sequence:

$$\sigma^n = \text{next}^n(\bot_\sigma) \tag{3.11}$$

for $n = 0, 1, \ldots$, where f^0 is the identity function and $f^{n+1} = f \circ f^n$.

In the general case we cannot assume the semilattice $D^\sharp(\sqsubseteq)$ to be a distributive semilattice of finite length, whence the state transition function may not be a complete join morphism. Under the weaker assumption that *next* is upper-continuous, the limit $next^\infty$ of the Kleene's sequence is defined as the least fixpoint point of *next*. This limit is computed by passing the functional $\lambda F.(next \circ F)$ as argument to the λ-calculus fixpoint combinator $FIX_{(\Sigma \overset{c}{\mapsto} \Sigma)}$:

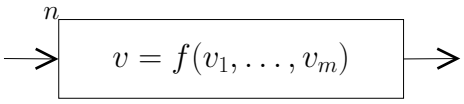
$$next^\infty = FIX_{(\Sigma \overset{c}{\mapsto} \Sigma)}(\lambda F.(next \circ F)) \tag{3.12}$$

where $FIX_D(f)$ denotes the least fixpoint of $f : D \overset{m}{\longmapsto} D$, Tarski [133].

3.3 Abstract Interpretation of Basic Program Units

An instance of the data-flow model to a particular program is built from a set of nodes $\mathcal{N}$ and a set of edges which correspond to the set of program labels $\mathcal{L} \in (\mathcal{N} \mapsto \wp(\mathcal{N}))$. We assume that the resulting model instance is a connected directed graph $\langle \mathcal{N}, \mathcal{L} \rangle$. In [32], the following elementary program units were defined by Cousot: a single *entry* node, *exit* nodes, *assignment* nodes, *test* nodes, simple *junction* nodes and *loop* junction nodes. As required by Kleene's first recursion theorem, the evaluation of expressions that can change the environment and test nodes have no side effects.

The abstract interpretation of basic program units is given by an interpretation $I^\sharp$. Given any assignment or test node $n \in \mathcal{N}$ and an input state vector $\sigma \in \Sigma$, $I^\sharp$ returns an output context $I^\sharp(n, \sigma')$ or two different output contexts when n is a test node. Since any state vector contains all the labelled abstract environments, each one identified by some node n, the consistency of $I^\sharp$ with respect to the *collecting* interpretation I, which is given by $I \subseteq \langle \alpha, \gamma \rangle I^\sharp$, imposes *local consistency* of the interpretations I and $I^\sharp$ at the level of basic language constructs. Let N_a be the set of assignment nodes. Then, for all $n \in N_a$, an assignment is of the form:



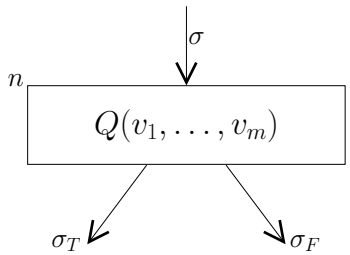
where $(v, v_1, \ldots, v_2) \in \mathcal{V}^{m+1}$ and $f(v_1, \ldots, v_m)$ is an expression of the language depending on the variables $v, v_1, \ldots, v_2$. Let ρ be the abstract environment found in the state vector $\sigma \in \Sigma$ at the program node n. Let $\vec{n}$ be the outgoing node of the assignment node n. We use the square bracket notation $map[k \mapsto e]$ to denote the update of the contents of a *map* at the key k with the new element e. For a program states map (σ) and for the an abstract environment map (ρ), it follows that:

$$\forall \sigma \in \Sigma, \quad \forall i \in \mathcal{V}, \quad i \neq v \implies I^\sharp(n, \sigma)(i) = \sigma \quad \text{and} \tag{3.13}$$

$$\forall \sigma \in \Sigma, \quad I^\sharp(n, \sigma)(v) = \sigma \left[\vec{n} \mapsto \rho \left[(v \mapsto \alpha \left(\{ f(v_1, \ldots, v_m) \mid (v_1, \ldots, v_m) \in \gamma\left(\rho(v_1)\right) \times \cdots \times \gamma(\rho(v_m)) \} \right) \right] \right] \tag{3.14}$$

The absence of side effects in the abstract interpretation of the expression $f(v_1, \ldots, v_2)$ is expressed by Def. (3.13). The local consistency of $I^\sharp$ is expressed by Def. (3.14): the abstract value $p^\sharp$ of v in the output environment is the abstraction of the set of values of the expression $f(v_1, \ldots, v_2)$ when the values $(v_1, \ldots, v_2)$ are chosen from the input abstract context ρ such that $\rho' = \rho[v \mapsto p^\sharp]$; finally, the output state vector σ' is updated with the output abstract environment at the program label $\vec{n}$, such that $\sigma' = \sigma[\vec{n} \mapsto \rho']$.

Let N_t be the set of test nodes. Then, for all $n \in N_t$ and given an input state vector $\sigma \in \Sigma$, the abstract interpretation $I^\sharp(n, \sigma)$ results of two output state vectors σ_T and σ_F associated with the "true" and "false" edge respectively:



where $Q(v_1, \ldots, v_m)$ is a boolean expression without side-effects depending on the variables $v_1, \ldots, v_m$. Let n_T and n_F be the program labels of the outgoing nodes of the test node $n \in N_t$. Also let ρ be the abstract environment found in the input state vector σ at the label n. Then, we define $I^\sharp(n, \sigma) = (\sigma_T, \sigma_F)$ such that for all $v \in \mathcal{V}$ we have:

$$\sigma_T(v) = \sigma\,[\, n_T \mapsto \rho\,[\, v \mapsto \alpha(\{t \mid t \in \gamma(\rho(v)) \wedge (\exists(v_1, \ldots, v_m) \in \gamma(\rho(v_1)) \times \cdots \times \gamma(\rho(v_m)) \mid Q(v_1, \ldots, v_2))\})\,]\,] \tag{3.15}$$

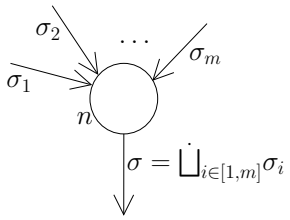
$$\sigma_F(v) = \sigma\,[\, n_F \mapsto \rho\,[\, v \mapsto \alpha(\{t \mid t \in \gamma(\rho(v)) \wedge (\exists(v_1, \ldots, v_m) \in \gamma(\rho(v_1)) \times \cdots \times \gamma(\rho(v_m)) \mid \neg Q(v_1, \ldots, v_2))\})\,]\,] \tag{3.16}$$

In this way we achieve a path-sensitive data-flow analysis where on the "true" edge the abstract value of a variable v is the abstraction of the set of values t chosen in the input context ρ, for which the evaluation of the predicate Q may yield the boolean value *True*. In the converse path, the state vector contains the abstract value of the variable v when the predicate Q yields to *False*.

As already mentioned, an abstract interpretation $I^\sharp$ is the least fixpoint solution of a system of recursive data-flow equations. In practice, these equations are iteratively evaluated according to an order of information propagation, in terms of program labels $\mathcal{N}$, across the basic program units. To this end, the transitive closure of the state transition function is used. Therefore, an abstract interpretation amounts to the computation of the limits of Kleene's sequences along all paths allowed on the program.

There are two possible solutions to the system of recursive equations [98]. The *merge over all path* (MOP) computes all possible functional compositions of the state transition function along all program paths. For each node $n \in \mathcal{N}$ inside a path, the transition function is evaluated for every input state and the resulting outputs are combined with the least upper bound operator $\bigsqcup$. In general there is an infinite number of program paths, which makes the MOP solution not computable. The solution to this problem is to compute an approximation of the MOP optimal solution called *minimum fixed point* (MFP). In this case, the least upper bound of the incoming states is computed before applying the *next* state transition function.

To demonstrate the computation of the MFP solution, we next describe the abstract interpretation of an execution path that eventually follows from a test node. From the test node, two execution paths are created, each one to be evaluated in pseudo-parallel until reaching an exit node, in which case the execution of the path ends, or a junction node, in which case pseudo-parallel execution paths are synchronized. As describe before, in order to compute the output state vector of a junction node, we must first compute the least upper bound of the input state vectors of the incoming edges that may be reached by an execution path. For a simple junction node n, we combine all input state vectors in point-wise form for the program labels $\{1, 2, \ldots, m\}$:



The limit of a Kleene's execution sequence in the abstract domain is computed by a transitive closure that traverses the direct graph $(\mathcal{N}, \mathcal{L})$ and applies the state transformation function *next* to each of the different types of nodes. Such transformation function specifies the state(s) for the outgoing edges of the node, in terms of the state(s) associated with the incoming edges to the node. The algorithm recursively apply functional applications of the state transformation function until all abstract environments ρ inside a state vector σ stabilizes with respect $\dot{\bigsqcup}$ (the point-wise version of the least upper bound $\bigsqcup$). The proof of the termination of this algorithm comes from the fact that, in one hand, sequences of state vectors form a strictly ascending chain, possibly after using widening/narrowing operators. Moreover, we assume that every loop contains a junction node.

The specification of an order of information propagation may lead to the instantiation of an optimal transitive closure algorithm. In [20], the notion of *weak topological order* (w.t.o.) is defined in terms of a total dominance order $\preceq$ that exists between the basic program units of a particular program, according to the information contained in the set of program labels $\mathcal{N}$. Let $i \to j$ denote an edge where is possible to jump from point i to point j by executing a single program step. An edge $u \to v$ such that $v \preceq u$ is called a *feedback edge*. For example, a program loop is a well-parenthesized component in the w.t.o. by having the program point v as its *head* and containing the program point u.

The main advantage of dominance order is that any sequential algorithm *à la* Gauss-Seidel [36] can be used to compute the limits of Kleene sequences. Such algorithms are called *chaotic iteration algorithms* and differ in the particular choice of the order in which the data-flow equations are applied. In [20] are enumerated two *iteration strategies*: (1) the *iterative* strategy simply applies the equations in sequence and stabilizes outermost components; (2) the *recursive* strategy recursively stabilizes the subcomponents of every component every time the component is stabilized. Of major relevance in our work is the fact that with the recursive strategy, *the stabilization of a component of a w.t.o. can be detected by stabilization of its head.*

3.4 Galois Connections

We assume that the *concrete* program properties are described by elements of a given set $P^\natural$. Let $\sqsubseteq^\natural$ be a partial order relation on $P^\natural$ defining the relative precision of concrete properties:

$p_1 \sqsubseteq^\natural p_2$ are comparable properties of the program, p_1 being more precise than p_2, the relative precision being left unquantified. The *abstract* program properties are assumed to be represented by elements of a poset $P^\sharp(\sqsubseteq^\sharp)$, where $(\sqsubseteq^\sharp)$ defines the relative precision of abstract properties.

Summarizing Section 3.1, the semantics of the abstract properties is given by the concretization function $\gamma \in P^\sharp \mapsto P^\natural$ such that $\gamma(p^\sharp)$ is the concrete property corresponding to the abstract description $p^\sharp \in P^\sharp$. The notion of approximation is formalized by the abstraction function $\alpha \in P^\natural \mapsto P^\sharp$ giving the best approximation $\alpha(p^\natural)$ of concrete properties $p^\natural \in P^\natural$.

If $p_1^\sharp = \alpha(p^\natural)$ and $p_1^\sharp \sqsubseteq^\sharp p_2^\sharp$ then $p_2^\sharp$ is a less precise abstract approximation of the concrete property $p^\natural$. Hence, the soundness of approximations, i.e. the fact that $p^\sharp$ is a valid approximation of the information given by $p^\natural$ can be expressed by $\alpha(p^\natural) \sqsubseteq^\sharp p^\sharp$. If $p_1^\natural = \gamma(p^\sharp)$ and $p_2^\natural \sqsubseteq^\natural p_1^\natural$ then $p^\sharp$ is also a correct approximation of the concrete property $p_2^\natural$ although this concrete property $p_2^\natural$ provides more accurate information about program executions than $p_1^\natural$. So the soundness of approximations, i.e. the fact that $p^\sharp$ is a valid approximation of the information given by $p^\natural$, can also be expressed by $p^\natural \sqsubseteq \gamma(p^\sharp)$.

Given the two posets $P^\natural(\sqsubseteq^\natural)$ and $P^\sharp(\sqsubseteq^\sharp)$, a *Galois connection* is written as:

$$P^\natural(\sqsubseteq^\natural) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} P^\sharp(\sqsubseteq^\sharp)$$

such that:

$$\forall p^\natural \in P^\natural : \forall p^\sharp \in P^\sharp : \alpha(p^\natural) \sqsubseteq^\sharp p^\sharp \Leftrightarrow p^\natural \sqsubseteq^\natural \gamma(p^\sharp) \tag{3.17}$$

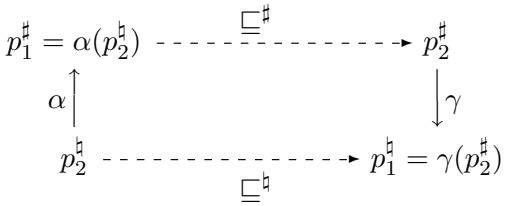


Figure 3.1: Galois connection commutative diagram

The loss of information in the abstraction process is sound, i.e. $\forall p^\natural \in P^\natural$, $\alpha(p^\natural)$ is an over-approximation of $p^\natural$. Let $p_1^\sharp = p^\sharp = \alpha(p^\natural) = \alpha(p_2^\natural)$. By reflexivity, $\alpha(p^\natural) \sqsubseteq^\sharp \alpha(p^\natural)$. Hence, the commutative diagram in Figure 3.1 induced by (3.5) and (3.17) shows that $\gamma \circ \alpha$ is *extensive* (from an abstract point of view, $\alpha(p^\natural)$ is as precise as possible):

$$\forall p^\natural \in P^\natural : p^\natural \sqsubseteq^\natural \gamma \circ \alpha(p^\natural) \tag{3.18}$$

In the same way, the concretization function γ introduces no loss of information (3.4). Let $p_1^\natural = p^\natural = \gamma(p^\sharp) = \gamma(p_2^\sharp)$. Since by reflexivity $\gamma(p^\sharp) \sqsubseteq^\natural \gamma(p^\sharp)$, hence $\alpha \circ \gamma$ is *reductive*:

$$\forall p^\sharp \in P^\sharp : \alpha \circ \gamma(p^\sharp) \sqsubseteq^\sharp p^\sharp \tag{3.19}$$

It follows that the concretization function γ is also monotone. The diagram in Figure 3.1 shows that when $p_1^\sharp \sqsubseteq^\sharp p_2^\sharp$, the reduction (3.19) and transitivity imply that $\alpha \circ \gamma(p_1^\sharp) \sqsubseteq^\sharp p_2^\sharp$, whence $\gamma(p_1^\sharp) \sqsubseteq^\natural \gamma(p_2^\sharp)$. In the same way, α is also monotone since $p_2^\natural \sqsubseteq^\natural p_1^\natural$ implies $p_2^\natural \sqsubseteq^\natural \gamma \circ \alpha(p_1^\natural)$ by (3.18) and transitivity, whence $\alpha(p_2^\natural) \sqsubseteq^\sharp \alpha(p_1^\natural)$.

The definitions of extensive (3.18) and reductive (3.19) morphisms can be combined to obtain the proof of idempotence of $\gamma \circ \alpha$ and $\alpha \circ \gamma$. In the first case, for all $p^\natural \in P^\natural$ and $p^\sharp \in P^\sharp$, we have $\alpha \circ \gamma(p^\sharp) \sqsubseteq^\sharp p^\sharp$ by the definition of reductive morphism (3.19) whence by monotony $\gamma \circ \alpha \circ \gamma(p^\sharp) \sqsubseteq^\sharp \gamma(^\sharp)$. Moreover, letting $p^\natural = \gamma(p^\sharp)$ in the definition of extensive morphism (3.18), we have $\gamma(p^\sharp) \sqsubseteq^\natural \gamma \circ \alpha \circ \gamma(p^\sharp)$. By antisymmetry, we conclude that:

$$\forall p^\sharp \in P^\sharp : \gamma \circ \alpha \circ \gamma(p^\sharp) = \gamma(p^\sharp) \tag{3.20}$$

Conversely, by letting $p^\sharp = \alpha(p^\natural)$ for all $p^\natural \in P^\natural$ in the definition of reductive morphism (3.19) we have $\alpha \circ \gamma(\alpha(p^\natural)) \sqsubseteq^\sharp \alpha(p^\natural)$. Moreover, the definition of extensive morphism (3.18) implies that $p^\natural \sqsubseteq^\natural \gamma \circ \alpha(p^\natural)$ holds for all $p^\natural \in P^\natural$ whence by monotony of α we have $\alpha(p^\natural) \sqsubseteq^\natural \alpha \circ \gamma \circ \alpha(p^\natural)$. By antisymmetry, we conclude that:

$$\forall p^\natural \in P^\natural : \alpha \circ \gamma \circ \alpha(p^\natural) = \alpha(p^\natural) \tag{3.21}$$

Consequently, a Galois connection defines two closure operators: a *lower closure operator* $\alpha \circ \gamma$ that is monotone, reductive and idempotent; and a *upper closure operator* $\gamma \circ \alpha$ that is monotone, extensive and idempotent.

Idempotence is a property of a closure operator φ on a set S ($\varphi \in S \mapsto S$) such that given a subset $X \subseteq S : x \in X : \varphi(x) = \varphi(\varphi(x))$. For example, an upper closure operator maps elements of subsets $X \subseteq S$ to the most precise element x' of X ($\forall x \in X : x' \sqsubseteq \varphi(x)$). This means that the information loss by the abstract interpretation process is always the same during successive abstractions, provided that the same abstraction function is used.

An important consequence of the existence of closure operators is that one can reason about abstract interpretation using only $P^\natural$ and its image by the upper closure operator $\gamma \circ \alpha$, therefore avoiding the direct use of $P^\sharp$. Moreover, abstract interpretations on $P^\sharp$ can be inductively defined from concrete interpretations on $P^\natural$ using the properties of closure operators.

A Galois connection is fully determined by either one of the abstraction or concretization functions: $\forall p^\natural \in P^\natural : \alpha(p^\natural) = \sqcap^\sharp\{p^\sharp \mid p^\natural \sqsubseteq^\natural \gamma(p^\sharp)\}$. Conversely, $\forall p^\sharp \in P^\sharp : \gamma(p^\sharp) = \sqcup^\natural\{p^\natural \mid \alpha(p^\natural) \sqsubseteq^\sharp p^\sharp\}$. First assume that $p^\natural \sqsubseteq^\natural \gamma(p^\sharp)$. By monotony of α, we have $\alpha(p^\natural) \sqsubseteq^\sharp \alpha(\gamma(p^\sharp))$. Moreover, $\alpha \circ \gamma \sqsubseteq^\sharp \lambda p^\sharp \bullet p^\sharp$ by definition of a lower closure operator. Hence,

$\alpha(p^\natural) \sqsubseteq^\sharp \alpha(\gamma(p^\sharp)) \sqsubseteq^\sharp p^\sharp$ implies $\alpha(p^\natural) \sqsubseteq^\sharp p^\sharp$ as expected. The proof showing that $\alpha(p^\natural) \sqsubseteq^\sharp p^\sharp$ implies $p^\natural \sqsubseteq^\natural \gamma(p^\sharp)$ is analogous.

The unique and reciprocal definitions of the functions α and γ can also be expressed in an alternative formulation of a Galois connection called the *pair algebra* framework [15]. Given the Galois connection $P^\natural(\sqsubseteq^\natural) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} P^\sharp(\sqsubseteq^\sharp)$ we re-define (3.17) in point-free style as $(\alpha^\cup, \gamma) \in (\sqsubseteq^\natural \leftarrow \sqsubseteq^\sharp)$, by stating that $(\alpha^\cup * \sqsubseteq^\sharp) = (\sqsubseteq^\natural * \gamma)$, where $^\cup$ (pronounced "wok") denotes the converse operation on relations and $*$ denotes the composition of relations. The total functions α and γ are termed *upper* and *lower adjoints*, respectively.

Galois connections can be uniquely determined by a definition of an abstract function that is a complete join morphism $\forall X \subseteq P^\natural : \alpha(\bigvee^\natural X) = \bigvee^\sharp \{\alpha(x) \mid x \in X\}$, when $\bigvee^\sharp$ exits, or a definition of a concretization function that is a complete meet morphism $\forall X \in P^\sharp : \gamma(\bigwedge^\sharp X) = \bigwedge^\natural \{\gamma(x) \mid x \in X\}$, when $\bigwedge^\natural$ exists.

Let P and $P^\sharp$ be complete lattices. Galois connections can also be defined in terms of a *representation function* $\beta : P \mapsto P^\sharp$ that maps a concrete value $p \in P$ to the best property describing it in $P^\sharp$ [98]. The representation function is based on the notion of *soundness relation*:

$$R_\beta \in \wp(P \times P^\sharp) \tag{3.22}$$

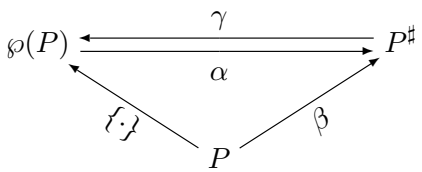
$\langle p, p^\sharp \rangle \in R_\beta$ means that the concrete semantics p of the program has the abstract property $p^\sharp$. Hence, the soundness relation (R_β) is defined from a given representation function (β) such that $p \, R_\beta \, p^\sharp \implies \beta(p) \sqsubseteq^\sharp p^\sharp$. Conversely, a representation function is defined from a soundness relation such that $\beta(p) = \bigsqcap \{p^\sharp \mid p \, R_\beta \, p^\sharp\}$.

The representation function gives rise to a Galois connection $P^\natural(\subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} P^\sharp(\sqsubseteq^\sharp)$ where $P^\natural = \wp(P)$ is the *collecting semantics* of concrete properties. Given a subset $X \subseteq P$ and an abstract property $p^\sharp \in P^\sharp$, the abstraction and concretization maps are defined by:

$$\alpha(X) = \bigsqcup \{\beta(x) \mid x \in X\} \tag{3.23}$$

$$\gamma(p^\sharp) = \{p \in P \mid \beta(p) \sqsubseteq^\sharp p^\sharp\} \tag{3.24}$$

It follows that $\alpha(\{p\}) = \beta(p)$ as is illustrated by the diagram:



3.5 Lifting Galois connections at Higher-Order

The Galois connection $P^\natural(\sqsubseteq^\natural) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} P^\sharp(\sqsubseteq^\sharp)$ defining sets of properties can be lifted at higher order to define sets of approximate monotone property transformers [28]:

$$P^\natural \overset{m}{\longmapsto} P^\natural(\dot{\sqsubseteq}^\natural) \underset{\lambda\varphi\bullet\alpha\circ\varphi\circ\gamma}{\overset{\lambda\phi\bullet\gamma\circ\phi\circ\alpha}{\leftrightarrows}} P^\sharp \overset{m}{\longmapsto} P^\sharp(\dot{\sqsubseteq}^\sharp) \tag{3.25}$$

where the ordering on functions is pointwise, that is $\varphi \mathrel{\dot{\sqsubseteq}} \phi$ if and only if $\varphi(x) \sqsubseteq \phi(x)$. Starting with an abstract property $x \in P^\sharp$, or its equivalent $\gamma(x) \in P^\natural$, the abstract properties transformer ϕ provide an over-approximation $\phi(x)$ of the property $\varphi(\gamma(x))$, obtained when the concrete properties transformer is applied to $\gamma(x)$. It follows that the choice of an approximation of program properties uniquely determines the way of approximating fixpoints of properties transformers. Figure 3.2 illustrates the higher-order Galois connection:

$$\begin{array}{ccc} x \in P^\sharp(\sqsubseteq^\sharp) & \xrightarrow{\phi} & P^\sharp(\sqsubseteq^\sharp) \\ \gamma\downarrow \uparrow\alpha & & \gamma\downarrow \uparrow\alpha \\ y \in P^\natural(\sqsubseteq^\natural) & \xrightarrow{\varphi} & P^\natural(\sqsubseteq^\natural) \end{array}$$

Figure 3.2: Commutative diagram of an higher-order Galois connection

The soundness requirement of the commutative diagram is $\varphi \circ \gamma(x) \sqsubseteq^\natural \gamma \circ \phi(x)$. Using the pointwise ordering, the soundness requirement implies that $\alpha \circ \varphi \circ \gamma \mathrel{\dot{\sqsubseteq}}^\sharp \phi$. Reciprocally for $y \in P^\natural$, the soundness requirement specifies that $\phi \circ \alpha(y) \sqsupseteq^\sharp \alpha \circ \varphi(y)$. Since the abstraction function uniquely determines the concretization by (3.2), we have by pointwise ordering that $\gamma \circ \phi \circ \alpha \mathrel{\dot{\sqsupseteq}}^\natural \varphi$.

3.6 Fixpoint Induction Using Galois Connections

Assume that $P^\natural(\sqsubseteq^\natural, \bigsqcup^\natural)$ and $P^\sharp(\sqsubseteq^\sharp, \bigsqcup^\sharp)$ are posets and that $F^\natural \in P^\natural \mapsto P^\natural$ provides the concrete semantics $lfp\, P^\natural$ of a program. The purpose of fixpoint induction using the Galois connection $P^\natural(\sqsubseteq^\natural) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} P^\sharp(\sqsubseteq^\sharp)$ is to obtained a definition for $\alpha(lfp\, P^\natural)$ [36].

The least fixpoint $lfp\, P^\natural$ is obtained as the limit of the iteration sequence $F^{\natural 0}(\bot^\natural) = \bot^\natural, \cdots, F^{\natural n+1}(\bot^\natural) = F(F^{\natural n}), \cdots, F^{\natural\omega} = \bigsqcup^\natural_{n\geqslant 0} F^{\natural n}(\bot^\natural) = lfp\, F^\natural$. Let $\bot^\sharp$ be an abstract infimum, $F^\sharp$ an abstract operator and $\sqcup^\sharp$ an abstract least upper bound. As an induction hypotheses, let $F^{\sharp n}(\bot^\sharp) = \alpha(F^{\natural n}(\bot^\natural))$ for all $n = 0, 1, \ldots \omega$. The following values for n are considered:

1. For $n = 0$, $\alpha(\bot^\natural) = \bot^\sharp$;

2. For $n \geqslant 0$, the induction hypothesis $\alpha(F^{\natural n}(\bot^\natural)) = F^{\sharp n}(\bot^\sharp)$ implies that $\alpha(F^{\natural n+1}(\bot^\natural)) = F^{\sharp n+1}(\bot^\sharp)$, which by definition of iteration sequences, means that $\alpha(F(F^{\natural n}(\bot^\natural))) = F(F^{\sharp n}(\bot^\sharp))$. Again by induction hypothesis, $\alpha(F(F^{\natural n}(\bot^\natural))) = F^\sharp(\alpha(F^{\natural n}(\bot^\natural)))$, which holds by the soundness requirement $F^\natural = \alpha \circ F^\natural \circ \gamma$ and when $\gamma \circ \alpha$ is extensive, implying that $\alpha \circ F^\natural = F^\sharp \circ \alpha$.

3. Finally for $n = \omega$, the induction hypothesis implies that $\alpha(\bigsqcup^\natural_{n \geqslant 0} F^{\natural n}(\bot^\natural)) = \bigsqcup^\sharp_{n \geqslant 0} F^{\sharp n}(\bot^\sharp)$. On the other hand, since α is a complete join morphism $\bigsqcup^\sharp_{n \geqslant 0} \alpha(F^{\natural n}(\bot^\natural)) = \alpha(\bigsqcup^\natural_{n \geqslant 0} F^{\natural n}(\bot^\natural))$, whence by transitivity we conclude that $\forall n \geqslant 0 : \alpha(F^{\natural n}(\bot^\natural)) = F^{\sharp n}(\bot^\sharp)$.

It follows that the least fixpoint $lfp\, F^\sharp$ of $F^\sharp \in P^\sharp(\sqsubseteq^\sharp) \stackrel{m}{\longmapsto} P^\sharp(\sqsubseteq^\sharp)$ is equal to $\bigsqcup^\sharp_{n \geqslant 0} F^{\sharp n}(\bot^\sharp)$. In fact, if $p^\sharp$ is a fixpoint of $P^\sharp$ such that $\bot^\sharp \sqsubseteq^\sharp p^\sharp$ then $F^{\sharp 0}(\bot^\sharp) = \bot^\sharp \sqsubseteq^\sharp p^\sharp$. Otherwise for $n \geqslant 0$, $F^{\sharp n}(\bot^\sharp) \sqsubseteq^\sharp p^\sharp$ implies that $F^{\natural n+1}(\bot^\sharp) = F^\sharp(F^{\sharp n}(\bot^\sharp)) \sqsubseteq^\sharp F^{\sharp n}(p^\sharp) = p^\sharp$ by definition of least upper bounds.

3.7 Fixpoint Abstraction Using Galois Connections

In general, the fixpoint inducing $\forall n \geqslant 0 : \alpha(F^{\natural n}(\bot^\natural)) = F^{\sharp n}(\bot^\sharp)$ is not computable. Hence, one must be satisfied with an abstract approximation $p^\sharp$ such that $\alpha(lfp F^\natural) \sqsubseteq^\sharp p^\sharp$. This concrete fixpoint approximation can be achieved using the higher-order Galois connection defined in (3.25), provided the Galois connection $P^\natural(\sqsubseteq^\natural) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} P^\sharp(\sqsubseteq^\sharp)$.

Let $p^\sharp = lfp\, \alpha \circ F^\natural \circ \gamma$ be the least fixpoint given by the Tarski's fixpoint theorem [133]. Since $p^\sharp$ is the least fixpoint, we have $\alpha \circ F^\natural \circ \gamma(p^\sharp) = p^\sharp$ whence $F^\natural \circ \gamma(p^\sharp) \sqsubseteq^\natural \gamma(p^\sharp)$. It follows that $\gamma(p^\sharp)$ is a post-fixpoint of $F^\natural$ whence $lfp\ F^\natural \sqsubseteq^\natural \gamma(p^\sharp)$ by Tarski's fixpoint theorem, or equivalently, $\alpha(lfp F^\natural) \sqsubseteq^\sharp p^\sharp = lfp\, \alpha \circ F^\natural \circ \gamma$.

A consequence of this fixpoint abstraction is that the choice of concrete semantics $F^\natural$ and the Galois connection $P^\natural(\sqsubseteq^\natural) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} P^\sharp(\sqsubseteq^\sharp)$ entirely determines the abstract semantics $lfp\, \alpha \circ F^\natural \circ \gamma$. Therefore, the abstract semantics $F^\sharp$ can be constructively derived from the concrete semantics by a formal computation consisting in calculating $\alpha \circ F^\natural \circ \gamma$ so that it uses operators on abstract properties only. This calculation is based on the fact that $\alpha \circ F^\natural \circ \gamma$ can be approximated by above by $F^\sharp$ such that $\forall p^\sharp \in P^\sharp : \alpha \circ F^\natural \circ \gamma(p^\sharp) \sqsubseteq^\sharp F^\sharp(p^\sharp)$.

Chapter 4

Worst-Case Execution Time

The present chapter identifies the several components that are typically used to build a modular and generic *worst-case execution time* (WCET) analyzer. These components are generically categorized into *program flow analysis*, *microarchitectural analysis* and *path analysis*. In one way or another, all the existent tools for WCET analysis require the information produced by each one of these components, but the techniques used to compute the information they provide can be quite different and the toolchain design that combines the several components may have many different configurations.

Starting by giving an introduction to the problem of estimating the WCET, we describe formal methods that can be used to compute sound WCET estimates, in particular those based on static analysis by abstract interpretation. This chapter gives an overview of the concepts applied in the design and formal definition of our WCET analyzer described in Chapter 6.

WCET estimates are of great importance in the development of embedded real-time systems since they are the main criteria used to perform schedulability analysis. Its purpose is to provide *a priori* information about the worst possible execution time of a given program before executing it in a system. The demand for such *safety* information is extremely relevant in hard real-time systems where the risk of failure caused by timing violations may endanger human life or put at risk substantial economic values.

Consider the possibility to use measurement-based methods in order to determine, of all possible running times, the "actual WCET", or the dual "actual BCET" (*best case execution time*). The inconvenient of measurement-based methods is that the complexity of the process, associated to variation of execution times according to all possible combinations of input data, can hardly be reduced without introducing safety problems. The reason is the input data that actually causes the worst-case execution time may be difficult, or even impossible, to predict.

For example, a function with three 16-bit integer arguments will have $2^{16} * 2^{16} * 2^{16} = 2^{48} \approx 3 \times 10^{14}$ potential executions. Additionally, pipelined processors would also introduce a variable number of execution times depending on the number of possible hardware states. Therefore, a safe (total) WCET measurement would not be feasibly computable. For this reason, lower and upper bounds are required for the BCET and WCET respectively, as illustrated in Fig. 4.1. The objective of WCET analysis is to find out an approximation of these bounds without introducing undue pessimism.

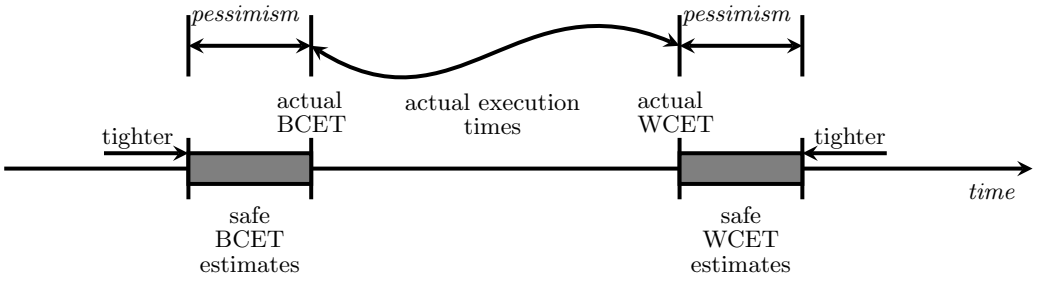


Figure 4.1: Relation between WCET, BCET, and possible program execution times

The focus on embedded systems determines which types of hardware we consider and the acceptable limitations regarding program flow and structure. In fact, the WCET depends both on the program flow of the source code, such as loop iterations and function calls, and on hardware factors, such as caches and pipelines. The analysis of the hardware is essential to improve the precision of WCET estimates. Nonetheless, these estimates should also be tight. In summary, soundness is provided by employing static analysis methods by abstract interpretation to exclude underestimation at hardware level. Additionally, tight program flow information at source level is essential to include as little overestimation as possible.

For these reasons, WCET estimation is a complex process and any WCET tool must provide solutions to several key problems in WCET analysis, including the representation and analysis of the control flow of programs, the modeling of the behavior and timing of hardware components such as pipelines and caches and, finally, the integration of control flow information and low-level timing information in order to obtain a safe and tight WCET estimate. Moreover, the WCET tool should ideally be integrated in the programmer's development environment, using namely the compiler infrastructure, to enable the use of the WCET as an explicit parameter from the programming and verification point of view.

In industrial contexts, where correctness is required by safety and criticality, the estimation of the WCET requires rigorous formal methods that should be directly applicable and economically affordable. The most common approaches to WCET analysis are *software model checking* and *program static verification*. The main characteristic of model checkers like UppAal [76], Kronos [21] and SPIN [68], is the existence of a model of a particular program and a specification, both used to perform multi-purpose formal verifications. The

main drawbacks are the difficulty to provide sensible temporal specifications and the state explosion in model instances.

On the other hand, the mechanisms of static analyzers like `aiT` [49], are defined for all programs written in a particular language and employ abstractions that are not specific to a particular program to analyze. They prove program properties by effectively computing an abstract semantics of programs expressed in fixpoint or constraint form. The abstract information can be used to support transformation and verification of programs. Our approach to static analysis in based on the theoretical foundations of Abstract Interpretation [30], which is a programming-language independent framework that allows the design of a wide class of abstract properties.

As previously mentioned, WCET estimation incorporates three sub-problems: *program flow analysis*, *microarchitectural analysis* and *path analysis*. The task of program flow analysis is to determine an approximation of the possible flows through the program. The resulting information is about which functions get called, how many times loops iterate, the nesting of `if-then-else` statements, etc. The flow information can be calculated manually by entering *manual annotations* into the program [104], or representing the flow information separately [80, 142]. *Automatic flow analysis* can be used to obtain flow information using abstract interpretation [44], instruction-level simulation [83] or data flow analysis [61].

Microarchitectural analysis computes approximations of the times that the system takes to execute all possible sequences of instructions allowed in a machine program. This requires the modeling of the host processor system, including the analysis of all hardware features that can affect timing behavior. In general, this process is divided into two separate analyses, termed by *global low-level analysis* and *local low-level analysis*. These tasks are preceded by a *value analysis*, specific to the instruction set of the host system, which computes the range of abstract values for registers and memory addresses.

Global low-level analysis considers the timing effects of the hardware features that reach across the entire program, i.e. determines how global effects can affect the execution time, but are not sufficient to generate actual execution times (expressed in CPU cycles). Typically, global low-level analysis includes the analysis of instruction caches [47, 92, 106, 137, 79], supporting a variety of cache replacement policies [108] with distinct cache levels [59], and data caches [48, 50]. Since exact analysis is impossible in the general case, most approaches use abstract interpretation to produce approximate yet safe analysis.

Local low-level analysis handles timing effects that depend on the history of computation, that is, the effects that depend not only on the execution of a single instruction, but also on the instruction's immediate neighbors. In this context, the analysis of pipeline overlaps is, naturally, the fundamental analysis. Nonetheless, some embedded systems may require the analysis of memory access speed in the presence of on-chip ROM and RAM memories, which are faster than off-chip memories.

Traditional approaches to *pipeline analysis* determine pipeline states for two instructions in isolation and then analyze the overlap resulting from concatenating them [81, 102]. Abstract interpretation approaches define an "abstract pipeline semantics" that describes all possible executions of the instructions in a safe way [122]. Other approaches use a generic processor simulator to run fragments of code and find local speed-up effects by generating concrete execution times. A common aspect in the mentioned approaches is the use of the information provided by the global low-level analysis, commonly referred to as *execution facts*.

Path analysis can also be termed *calculation* and is responsible for combining the results obtained with the *program flow analysis* and the *global* and *local low-level analysis* with the final purpose to calculate the WCET estimate for the program. The most popular method used to model and calculate the optimization of the worst-case execution time problem is *integer linear programming* (ILP). This technique was introduced in the context of WCET analysis by [105] and was latter developed in [80, 102, 130]. ILP expresses program flow and atomic execution times using algebraic constraints. The WCET, or the dual BCET, are defined in terms of an objective function, which is maximized or minimized, while satisfying all constraints.

"State of the art" tools for WCET analysis, prominently AbsInt's `aiT` [49], perform the microarchitectural analysis by computing approximations, using abstract interpretation (AI), of the times that the system takes to execute all possible sequences of instructions allowed in a machine program. This requires the abstract modeling of the host processor system, which most commonly include the abstract semantics of the processor instruction set, the abstract semantics of cache memories and the timing model of processor pipelines. The program flow analysis is mainly accomplished by user manual annotations and the path analysis estimated the WCET using ILP. A comparison between the combined approach of AI+ILP with model checking approaches is provided by Reinhard Wilhelm in [142].

The AI+ILP approach is modular, in the sense that it separates processor behaviour prediction and worst-case path determination. The AI component used to predict the processor behaviour uses a complex, but precise, fixpoint computation. The remaining ILP modeling the control flow of the program is usually of moderate size and solvable quickly. On the other hand, model checkers are typically monolithic tools unable to offer acceptable performance.

There are also some methods that integrate the several steps into a single algorithm. For example, [83] uses a modified CPU simulator to simultaneously perform program flow, cache and pipeline analysis, altogether with calculation. In comparison to tools that keep the corresponding algorithms separate, this approach is able to produce more precise results, because it excludes infeasible paths, but it is less efficient and relies in a very sophisticated and detailed CPU modeling.

Chapter 5

Generic Data Flow Framework

This chapter introduces a programming language-independent and generic data flow framework used for computing parametrizable abstract interpretations and introduces the corresponding declarative definitions using Haskell as the host programming language. The data-flow equations of a program are instantiated according to the data dependencies found on the program at compile-time, modelled according to the *weak topological order* of the program [20]. In this way, and in order to give a compositional characterization to the fixpoint semantics, the order of the data-flow equations inside a Kleene increasing chain [74] is obtained by induction on the program syntactic structure. Therefore, the instantiation of an abstract interpreter corresponds to a particular fixpoint algorithm, which involves the choice of a chaotic iteration strategy [36] to efficiently compute the Kleene *sequences*.

We formulate a constructive fixpoint semantics based on expressions of a two-level denotational meta-language aiming at compositionality in the value domain. The main advantage is the possibility to compile type-safe fixpoint interpreters automatically, and in a flexible way, for a variety of control flow patterns in machine programs. Denotational definitions are factored in two stages, which is equivalent to the definition of a *core semantics* at compile-time and an *abstract interpretation* at run-time. Supported by the *compositionality assumption* of Stoy [132], the core semantics expresses control flows by means of higher-order relational combinators of the run-time entities.

Rice's theorem is an important result for analysis and verification of programs and can be informally paraphrased by stating that all interesting questions about the behavior of programs are *undecidable*. Examples of such questions are: "what is the possible output?"; "will the value of `x` be read in the future?"; "is the value of the integer variable `x` always positive?"; "what is a lower and upper bound on the value of the integer variable `x`?" Therefore, the focus of static analysis is not to decide such properties but rather to provide *approximative* answers that are still precise enough to allow the verification of such approximate properties. However, such approximations must be *sound*, in the sense that all allowed run-time values must be included in the approximation statically computed at compile-time.

Data flow analysis [72, 98], also called *monotone frameworks*, is a process characterized by the use of some compile-time representation of the program, most commonly its *control flow graph* (CFG), and an algebraic structure, typically a lattice, describing the "approximate" values of interest for the analysis. For each programming language construct, it is defined a *data flow constraint* that relates the value of a program variable across all the nodes defined in the CFG. For a complete CFG, a collection of constraints over the program variables can be systematically extracted by defining all the constraints as equations or inequations with monotone right-hand sides, which are then given as input to a fixpoint algorithm that computes the unique *least fixpoint solution*. Typical examples of data flow analysis are *liveness*, *available expressions*, *reaching definitions*, *sign analysis* and *constant propagation* [124].

5.1 Fixpoint Semantics

This section describes an application of the hierarchy of fixpoint semantics of a transition system by abstract interpretation proposed by Cousot in [27], in conjunction with the notion of *weak topological order* [20], which establishes a dependency order among syntactic phrases, aiming at defining a more compact program semantics that we have designated by *meta-trace semantics*. The advantages of *meta-trace semantics* are the ability to define efficient fixpoint algorithms, by means of chaotic program-specific iterations strategies, and the ability to define proper syntactic representations in denotational interpretations in order to automatically generate compositional and generic data-flow analyzers. The contents of this section are the foundations for Contrib. (ii) and Contrib. (iii) referred in Section 1.

Proofs of convergence and termination of chaotic fixpoint iterations can be found in [36]. We start with a brief discussion on the subject and then describe some related work. Using a decomposition by partitioning, the fixpoint equation $X = F(X)$ can be decomposed into a system of equations:

$$\begin{cases} X_i = F_i(X_1, X_2, \ldots, X_n) \\ i = 1, \ldots, n \end{cases} \tag{5.1}$$

where each X_i belongs to a cpo or complete lattice $P_i(\sqsubseteq_i)$ and $F_i = (X_1, X_2, \ldots, X_n)$ is equal to the i-th component $F(X)[i]$ of $F(X)$. If F is upper-continuous then the least fixpoint $lfp\, F = \bigsqcup_{k \geqslant 0} F^k$ where $F^0 = \bot$ and $F^{k+1} = F(F^k)$ can be computed by Jacobi's method of successive approximations or by Gauss-Seidel's iterative method. In particular, the Gauss-Seidel method consists in continually re-injecting previous results in the fixpoint computations in order to accelerate the convergence.

Without sufficient hypothesis on F, Jacobi's method may converge while the Gauss-Seidel one diverges, and vice-versa. However, when F is upper-continuous, or monotone using transfinite iteration sequences [34], the convergence of any *chaotic iteration method* to the

least fixpoint of F is assured by determining, at each step, which are the components of the systems of equations which will evolve and in what order, as long as no component is forgotten indefinitely, so as to ensure fairness. Let J be a subset of $\{1,\ldots,n\}$. We denote by F_J the map defined by $F_J(X_1,\ldots,X_n) = \langle Y_1,\ldots,Y_n\rangle$ where, for all $i = 1,\ldots,n$:

$$\begin{cases} Y_i = F_i(X_1,\ldots,X_n) & \text{if } i \in J \\ Y_i = X_i & \text{if } i \notin J \end{cases} \tag{5.2}$$

An *ascending sequence of chaotic iterations* of F is a sequence X^k, $k \geqslant 0$ of vectors of $\prod_{i=1}^n P_i$, starting from the infimum $X^0 = \prod_{i=1}^n \bot_i$, and is recursively defined for $k > 0$ as $X^k = F_{J_{k-1}}(X^{k-1})$, where J_k, is a *weakly fair* sequence of subsets of $\{1,\ldots,n\}$, so that no component is forgotten indefinitely. The difference between the iteration methods of Jacobi and Gauss-Seidel is the choice of the successive values of k [33]. As the Cousots advocate in [28, 36], the compositional design of an abstract interpreter should involve the choice of a chaotic iteration strategy so as to mimic the actual program execution. An example of the definition of particular strategies of chaotic iterations, namely the *iterative* and the *recursive* strategies, is given in [20].

Two well-known generalizations are *asynchronous iterations* [25] and *minimal function graphs* [69]. The former approach computes fixpoints using a parallel implementation, where X is a shared array and each process i reads the value x_j of element $X[j]$ in any order for $j = 1,\ldots,n$. The iteration is performed by letting $x'_i = F_i(x_1,\ldots,x_n)$, whose result is finally asynchronously written in shared memory $X[i]$. The later approach defines systems of functional fixpoint equations $f_i(\vec{X_i}) = F_i[f_1,\ldots,f_n](\vec{X_i}), i = 1,\ldots,n$. Since the iteration strategy defines the order in which each value $f_i(\vec{X_i})$ is computed, it is necessary and sufficient to define their inputs as subsets ϕ_i of the domain P_i of $\vec{X_i}$. The returned output belongs to a subset of the domain of X called the ϕ-F-**closure** and such that $\phi_i \subseteq \phi\text{-}F\text{-}\mathbf{closure}_i \subseteq P_i$.

5.1.1 Declarative Approach

The *semantics of a program* provides a formal mathematical model of all possible behaviors of a computer system executing this program in interaction with any possible environment at some level of abstraction. For the purpose of static timing analysis, the automatic determination of timing program properties is accomplished by inspecting the source code of some assembly program, $P \in \text{Prog}$, that consists in a sequence of machine instructions, $I \in \text{Instr}$. Program properties are defined in the abstract domain $\mathbb{C}$, denoting all the possible abstract hardware states occurring in the target platform.

The analysis is based on notion of *labelled program states*, $\langle l, \sigma\rangle \in \Sigma$, that associate every *program label*, $l \in \text{Lab}$, to some *program invariant*, $\sigma \in \text{Invs}$, that soundly approximates the dynamic behavior of a program $P \in \text{Prog}$ [31].

$$\text{Instrs} \in \text{Prog} \mapsto \wp(\text{Instr}) \tag{5.3}$$
$$\text{Instrs}[\![I_1; \ldots; I_n]\!] \triangleq \{I_1, \ldots, I_n\}$$
$$\text{Invs} \in \text{Prog} \mapsto \wp(\text{Lab} \hookrightarrow \mathbb{C}) \tag{5.4}$$
$$\text{Invs}[\![P]\!] \triangleq \text{at}_P[\![P]\!] \mapsto \mathbb{C}$$
$$\Sigma \in \text{Prog} \mapsto \wp(\text{Lab} \hookrightarrow \text{Invs}) \tag{5.5}$$
$$\Sigma[\![P]\!] \triangleq \text{at}_P[\![P]\!] \mapsto \text{Invs}[\![P]\!]$$

The type of program invariants $\text{Invs}[\![P]\!]$ is the set of total maps from the program labels, l, to an *abstract environment*, $\rho \in \mathbb{C}$. The program invariants are defined in Haskell by the data type $(Invs\ a)$, where a is a polymorphic type variable denoting $\mathbb{C}$, as specified by Def. (5.4). In its turn, abstract environments, $\rho \in \mathbb{C}$, are denoted by the datatype $(\mathbf{Env}\ a)$, which keeps the *value* of some polymorphic abstract hardware state a denoting $\mathbb{C}$. Finally, program states are defined by the constructor $(\mathbf{St}\ a)$, which models the one-to-one relation between each *label* and the corresponding program invariants map *invs*, as specified by Def. (5.5). For sake of convenience, we use the record syntax of Haskell to define the constructor $(\mathbf{St}\ a)$ as a bijection from *Lab* to $(Invs\ a)$.

```
data Env a = Env {value :: a}
type Lab = Int
type Invs a = Map Lab (Env a)
data St a = St {label :: Lab, invs :: Invs a}
```

Let $I \in \text{Instrs}[\![P]\!]$ be a program instruction. Every label l inside a program is contained inside the set in_P, and identifies either the state "at" the beginning or the state just "after" an instruction I.

$$\text{at}_P, \text{after}_P \in \text{Instrs}[\![P]\!] \mapsto \text{Lab}$$
$$\text{in}_P \in \text{Instrs}[\![P]\!] \mapsto \wp(\text{Lab})$$
$$\text{in}_P[\![I]\!] \triangleq \{\text{at}_P[\![I]\!], \text{after}_P[\![I]\!]\} \tag{5.6}$$
$$\text{in}_P[\![I_1; \ldots; I_n]\!] \triangleq \bigcup_{i=1}^{n} \text{in}_P[\![I_i]\!] \tag{5.7}$$

As mentioned in Section 3.2, by using Def. (3.9), the program states of Def. (5.5) are proved to be uniquely identified by their labels. Hence, program states can be ordered according to the notion of *weak topological ordering* (w.t.o. for short) [20]. The objective is to represent the structure of programs isomorphically to a hierarchical order of program states using, for that purpose, a total order $\preceq$ on Lab. The elements inside matching parentheses are called *components* and the first element of a component is called the "head". For example, a program with four components could have the following w.t.o.:

$$(l_1^1 \ \cdots \ l_1^{n_1} \ (l_2^i \ \cdots \ l_2^{n_i} \ (l_3^j \ \cdots \ l_3^{n_j}) \ (l_2^u \ \cdots \ l_2^{n_u}))) \tag{5.8}$$

The labels pertaining to the component 1 are the sequence $l_1^1 \cdots l_1^{n_1}$, where the identifiers $1, \ldots, n_1$ (upper indices) define a set of labels sharing a sequential hierarchy belonging to the same component 1 (lower index). The second component has a first sequential order of labels, $l_2^i \cdots l_2^{n_i}$, starting with the identifier i and ending with n_i, then is interposed by a third component with a sequential hierarchy of $j \ldots n_j$, and, finally, is completed with the fourth sequential hierarchy of $u \ldots n_u$. The total order ($\preceq$) is induced by the position that each label identifier has in (5.8).

However, this by no way means that all instructions of the machine program are executed in sequence. On the contrary, the w.t.o. defines the program points in the machine program from where executions can "continue", "jump", "return", etc. The notion of *components* and *heads* of components defines precisely the control flow that can be extracted directly from the machine program, at compile time, e.g. by analyzing the "program counter" offset used by a branch instruction.

"Head" labels are defined as the labels in the first position of a component. For any given label l, the set of heads of the nested components containing l is denoted by $w(l)$. Assuming that ARM9 [125] is the target instruction set architecture, heads of components, like the ones upper-indexed by 1, i, j and u in (5.8), are necessarily either an entry point of a procedure (e.g. after a *branch-and-link* instruction, '`bl`'), the head of an intraprocedural loop (before a *conditional-branch* instruction, '`bgt`', '`beq`', etc.) or the hook point on the *caller* procedure after a procedure return (next instruction after a *branch-and-link*). The last labels inside a component, n_1, n_i, n_j and n_u in Def. (5.8), represent either the label before an intraprocedural loop, the next-to-last label of an intraprocedural loop or the return point of a procedure (after a *load-registers-and-return* instruction, '`ldmfd`', '`ldmfa`', etc.).

Using the hierarchical order ($\preceq$), the w.t.o. induces a dependency graph, which the set of vertices, denoted by the type *Lab*, are hierarchically ordered such that an edge $i \rightarrow j$ is defined if it is possible to jump from label i to label j by executing a single program step. An edge $i \rightarrow j$ such that $j \preceq i$ is called a *feedback edge*.

$$(i \prec j \wedge j \notin w(i)) \vee (j \preceq i \wedge j \in w(i)) \tag{5.9}$$

Example 2. The weak topological order of a simple program.
As an illustrating example of machine state labeling consider the source code in Fig. 5.1 and corresponding labelled machine program in Fig. 5.2.

Fig. 5.2 shows two *root* labels, 'root_0' and 'root_11', one for each procedure, 'main' and 'foo'. They label the states just before the instruction '`mov ip, sp`' which stores the intra-procedure pointer '`ip`' into the stack pointer '`sp`'. The procedure call is made through the *branch-and-link* instruction '`bl`', which starts with the *intra*-procedural sequential label 'n_5' and produces the target label 'call_11'.

The start label of the next instruction is a "hook" point ('hook_6'), stating that the execution

```
int main(void) {
  int y = foo (5);
  return y;
}

foo (x) {
  while (x>0) {
    x--;
  }
  return x;
}
```

Figure 5.1: Source code example

continues at this point after the return of the procedure call. The intra-procedural loop inside the procedure 'foo' produces the "head" (underlined) label 'head_22' that, in conjunction with the label 'n_17' (feedback edge), states that the instructions between the labels 17 and 22 will be recursively executed by the static analyzer until the conditional instruction `bgt -20` evaluates to "false" in the abstract domain. Finally, each procedure returns with a non-enumerable target label "exit" that can only be determined in function of the caller's "hook" point.

The corresponding weak topological order is:

$$(0 \cdots 5\ (11\ 12 \cdots 16\ 20\ \ldots (\underline{22}\ 17\ \cdots 21)\ 22 \cdots 25\ (6 \cdots 10))) \quad \blacktriangle \tag{5.10}$$

Since the position of labels must be identified inside the components of a w.t.o., they cannot be modelled using exclusively integer values. Instead, labels are now defined by the constructor **Label**. A label at the beginning of a procedure call is constructed using **Root**. Regular sequential labels are constructed using **Label**, "head" labels are constructed using **Head**. A label after a procedure call is constructed using **Call**, a label after a procedure return is constructed using **Exit** and, finally, the first label after a procedure return is constructed using **Hook**.

```
data Label = Empty
           | Root Identifier | Label Identifier | Head Identifier
           | Call Identifier | Exit Identifier  | Hook Identifier
```

The integer identifying each label is stored inside an **Identifier**, and accessed via the record function *labelId*, along with information related to the *procedure* that contains that label. Each procedure contains different "sections", for example '.L5' and '.L4' in Fig. 5.2, which correspond to the different basic blocks in the source code. Whence, the initial definition of program states, (**St** a) must now be redefined to associate the type **Label** to the field *label*.

```
data Identifier = Identifier {labelId :: Lab, procedure :: Proc}
data Proc = Proc {procId :: Int, section :: String, procName :: String}
data St a = St {label :: Label, invs :: Invs a}
```

```
                        n1: mov    ip , sp                 :root_0 {"main"};1
                        n2: stmfd  sp!, {fp,ip,lr,pc}      :n1
                        n3: sub    fp , ip , #4            :n2
                        n4: sub    sp , sp , #4            :n3
                        n5: mov    r0 , #5                 :n4
call_11 {("foo","main")}: bl       24                      :n5
                        n7: mov    r3 , r0                 :hook (6,"main")
                        n8: str    r3 , [fp , #-16]        :n7
                        n9: ldr    r3 , [fp , #-16]        :n8
                       n10: mov    r0 , r3                 :n9
            exit {"main"}: ldmfd  sp , {r3,fp,sp,pc}      :n10
                       n12: mov    ip , sp                 :root_11 {"foo"};2
                       n13: stmfd  sp!, {fp,ip,lr,pc}      :n12
                       n14: sub    fp , ip , #4            :n13
                       n15: sub    sp , sp , #4            :n14
                       n16: str    r0 , [fp , #-16]        :n15
                       n20: b      16                      :n16
                       n18: ldr    r3 , [fp , #-16]        :n17 {(".L5","foo")}
                       n19: sub    r3 , r3 , #1            :n18
                       n20: str    r3 , [fp , #-16]        :n19
                       n21: ldr    r3 , [fp , #-16]        :n20 {(".L4","foo")}
                       n22: cmp    r3 , #0                 :n21
                       n17: bgt    -20                     :head_22
                       n23: bgt    -20                     :n22
                       n24: ldr    r3 , [fp , #-16]        :n23
                       n25: mov    r0 , r3                 :n24
    exit {(".L4","foo")}: ldmfd  sp , {r3,fp,sp,pc}      :n25
```

Figure 5.2: Labelled instruction set for source program 5.1

The weak topological order is at the foundations of the *chaotic* fixpoint algorithm [20, 36]. It reflects the syntactical dependencies that, together with an iteration chaotic strategy, also specifies data dependencies in the data-flow analysis. The objective is to perform a flow-sensitive, path-sensitive and context-sensitive analysis of machine programs where the history of a computation can be taken into consideration, while supporting interprocedural analysis [128].

This is particularly relevant for *pipeline analysis*, for which the fixpoint algorithm provides an effective method for pipeline state transversal [142]. Put simply, the chaotic iteration strategy consists in recursively traversing the dependency graph induced by the program's w.t.o. Hence, chaotic fixpoint iterations, parametrized by an appropriate iteration strategy, are able to mimic the execution orders of the program [28].

Let G be any rooted dependency graph. G is denoted by a triple (N, E, r), where N is the set of its nodes, E the set of edges and r its root. A *path* p in G is a sequence of nodes in N $(n_1, \ldots, n_i, n_j, \ldots, n_k)$ such that $\forall (i,j), \exists n_i \rightarrow n_j \in E$ such that Def. (5.9) holds. A path p is said to lead from n_1 to n_k and can also be represented as the corresponding ordered set of edges $\langle E, \preceq \rangle$, where $E = \{(i,j) \mid i \prec j\} \cup \{(j,i) \mid j \preceq i\}$.

Fig. 5.2 illustrates how the machine program is labelled according to a weak topological order where, according to Def. (5.6), each instruction is surrounded by two labels. The desired behavior of the w.t.o. is to let the total order over labels to produce the same precedence order of instructions as would be obtained by simulation of the machine program. Intuitively, the representation in Fig. 5.2 uses a w.t.o. to convey the relational semantics of the machine program, which is defined by a nondeterministic transition system $\langle \Sigma[\![P]\!], \tau \rangle$, where ternary relations $\tau \subseteq (\Sigma[\![P]\!] \times \text{Instrs}[\![P]\!] \times \Sigma[\![P]\!])$ are defined between program states $\Sigma[\![P]\!]$ that are "connected" by the labelling process.

Given a syntactic object $\text{Instrs}[\![P]\!]$, an input-output relation is established between a state $\langle l, \sigma \rangle \in \Sigma[\![P]\!]$, which we abbreviate to $\sigma_l \in \Sigma[\![P]\!]$ for sake of simplicity, and its possible successors. Let a be a type variable for program states $\Sigma[\![P]\!]$. Then, input-output relations are defined in Haskell as:

```
data Rel a = Rel (a, Instr, a)
type RelSemantics a = [Rel a]
```

Therefore, for each procedure there exists an isomorphism between the dependency graph G and the labelled relational semantics $\langle \Sigma[\![P]\!], \tau \rangle$, where the set of all nodes, $\{r\} \cup N$, correspond to the labels of program states $\Sigma[\![P]\!]$ and the edges $E = (N \times N)$ correspond to the pairs of label identifiers present in the set of relations τ. Moreover, apart from the invariants stored in the program states $\Sigma[\![P]\!]$, the dependency graph G and the relational semantics $\langle \Sigma[\![P]\!], \tau \rangle$ are isomorphic structures, as Section 7.2.3 will explore more in detail.

The data-flow static analysis framework is defined to be a pair $\langle \Sigma[\![P]\!], F \rangle$, where F is a space of functions acting in $\Sigma[\![P]\!]$. To each edge (i, j) of G is associated a propagation function $f_{(i,j)} \in F$, which represents the change only at the component of $\Sigma[\![P]\!]$ at label j (for simplicity simply referred to as $\Sigma[j]$), that is, a change of relevant data inside the invariants map, $\text{Invs}[\![P]\!]$, as control passes from the start i, through i, to the start of j. Once the set $S = \{f_{(i,j)} : (i, j) \in E\}$ is given, the graph-dependent space F of propagation functions is defined as the smallest set of functions acting in $\mathbb{C}$, which contains S and the identity map, and which is closed under functional compositions and joins.

Hereby, it follows the general set of data-propagation equations, where for each $n_j \in N$, x_j denotes the data instance available at the start of n_j:

$$\begin{aligned} x_r &= \bot_{\mathbb{C}} \\ x_j &= \bigsqcup_{(i,j) \in E} f_{(i,j)}(x_i), n_j \in N - \{r\} \end{aligned} \tag{5.11}$$

These equations describe state transformations which are locally collected across adjacent basic blocks of the graph, starting with the "null" ($\bot_{\mathbb{C}}$) information at the program entry. The optimal solution of these equations is the *merge-over-all-paths* solution (MOP):

$$y_i = \bigsqcup\{f_p(\bot) : p \in \text{path}_G(r, j)\}, j \in N \tag{5.12}$$

where we define $f_p = f_{(i_k,j_k)} \circ f_{(i_{k-1},j_{k-1})} \circ \cdots \circ f_{(i_1,j_1)}$ for each path $p = (n_{i_1}, n_{j_1} \ldots, n_{i_k}, n_{j_k})$. If p does not exist, then f_p is defined to be the identity map on $\mathbb{C}$. Since the MOP solution is undecidable in general, an approximating iterative algorithm is required to yield the *minimal fixed point* (MFP) by computing joins at those program points that have multiple incoming edges before proceeding with functional application. Nonetheless, the use of functional application to compute fixpoint solutions is suitable for interprocedural analysis, where the semantics of procedure calls is modeled as the composition of structured program blocks, aiming to establish algebraic compositions of input-output relations for each of the procedure blocks.

Our *functional approach* to interprocedural analysis is supported on formalisms of weak topological order and the *least fixpoint* (in alternative to the *minimal fixed point*) of the relational semantics. Using the w.t.o., procedures are represented as components (5.8), where one can jump back to the original call site after a procedure call, interpreting procedure calls as "super-operations" whose effect on the abstract environment can be computed using the composition of the relations involved. To this end, the syntactic objects of relations are defined by means of inductive expression constructor **Expr**. Accordingly, the constructor of relations (**Rel** a) is re-defined to possible denote state transitions between a non-empty sequence of instructions instead of a single instruction only.

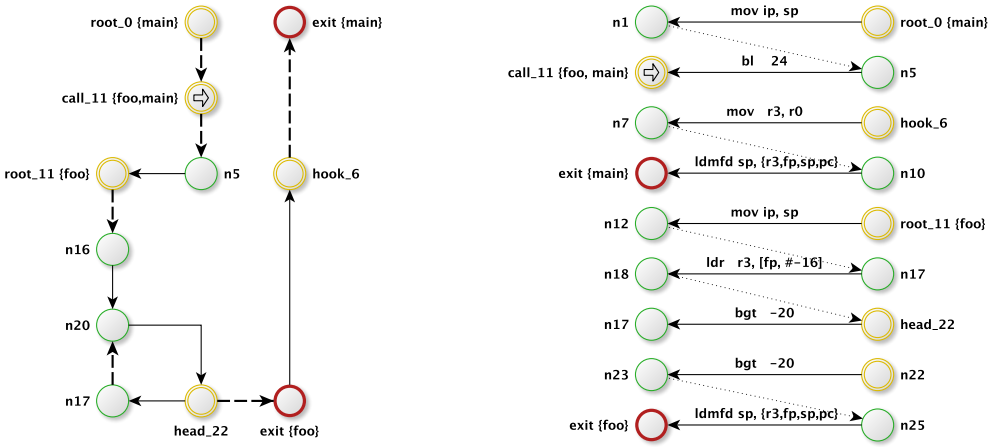
```
data Expr = Expr Instr
          | Cons Instr Expr
data Rel a = Rel (a, Expr, a)
```

Procedures are in general multi-exit blocks of code, which makes it impossible to know at compile time what the target "exit" label of the last transition relation inside the procedure block will be. However, instead of transforming a program with procedures into a procedure-less program, by including all the "root" labels in the same dependency graph, the design of the fixpoint algorithm keeps the formalism based on functional application and least upper bound operators, but dynamically determines the target label to be equal to the "hook" point of the call site. Section 6.5 describes how this mechanism enables a context-sensitive analysis by means of a procedure call stack.

Another mathematical model for expressing the semantics of a program is that of the *trace semantics* and it can be obtained as a refinement of the relational semantics [27, 30, 37]. From the observation of program execution, this is the most precise semantics that can be considered. Starting from an initial state, the trace semantics models the execution of a program, for a given specific interaction with its environment, as a sequence of states, observed at discrete intervals of time, moving from one state to the next state by executing an atomic program step. For WCET analysis purposes, we consider only terminating programs.

Therefore, the trace semantics always ends with a final regular state.

Although the relational semantics of Fig. 5.2 does not specify all the possible intermediate states that can be computed during program executions, it does specify the data dependencies which are known at compile time. Since program states are labelled, these data dependencies can be computed along fixpoint iterations by means of monotone propagation functions and least upper bound operators. Using Def. (5.11), the number of program states kept along fixpoint computation is equal to the number of program labels. On the contrary, the trace semantics is a concatenation of variable length of program states, observed at discrete intervals of time, computed during "actual" (run-time) program executions. Nevertheless, the labelled relational semantics in Fig. 5.3(b) provide an *abstraction* of the trace semantics and provides, at the same time, the basis for the definition of a *meta-trace semantics*, e.g. in Fig. 5.3(a), which is a more compact representation of the Kleene ascending chains that will be unfolded during fixpoint computation. In fact, the unfolded *meta-trace semantics* is semantically equivalent to the *trace semantics*.



(a) Meta-Trace Semantics

(b) Relational Semantics

Figure 5.3: Alternative semantic representations (Meta-Trace and Relational)

The meta-trace is a convenient representation because it accommodates in a single representation several aspects related to fixpoint semantics: (1) it expresses the weak topological order of the program (compare the w.t.o in (5.10) with the sequence of states in Fig. 5.3(a)); it provides a graph-based interpretation of the w.t.o. which, by means of functional application and joins, computes approximations to the MOP fixpoint solution (5.12); it conveys a way to apply the Jacobi's method [36] of successive approximations to compute the MFP using *chaotic iteration* strategies [20]; and it provides a way to reduce the number of comparison between elements of the domain of program states, which can be very useful when this test is computational heavy.

Example 3. Example of a chaotic fixpoint strategy.
For the machine code example of Fig. 5.2 described in Example 2, and taking into consideration the weak topological order (5.10), the respective recursive iteration strategy is the following:

$$0 \cdots 5\ 11 \cdots 16\ 20\ 21\ [\underline{22}\ 17\ \cdots 21]^*\ 22 \cdots 25\ 6 \cdots 10 \tag{5.13}$$

The label 22 is repeated after the [ ]* "iterate until stabilization" operator in order to provide a path-sensitive analysis. In this way, information dependent on the predicates at conditional branch instructions is taken into consideration. For instance, if a branch instruction represents a condition $x > 0$ in the source code, then it would assume that indeed $x > 0$ holds on the beginning of the target path of the branch (the head of the conditional component), and that $x <= 0$ holds on the fall-through path. ▲

The chaotic iteration strategy is chosen as a mimic (simulation) of actual program executions. At each step, the w.t.o. determine which are the components of the system of equations of Def. (5.11) that are updated with the effect produced by each data-propagation function and in what order those effects are computed. Since no label is indefinitely forgotten, the chaotic iteration method converges to the least fixpoint of the space of functions F. If F is upper-continuous [36], then the least fixpoint $lfp^{\sqsubseteq}_{\perp} F = \bigsqcup_{k \geqslant 0} F^k$ where $F^0 = \perp$ and $F^{k+1} = F(F^k)$ can be computed by the Jacobi's method of successive approximations.

In order to solve fixpoint equations like $\Sigma[\![P]\!] = F(\Sigma[\![P]\!])$, the data-flow equations defined in (5.11) are redefined in terms of the iteration k, where the invariant $\sigma_i = \Sigma[i]$ and the invariant $\sigma_j = \Sigma[j]$ are two invariant such that either $i \prec j$ or $j \preceq i$:

$$\sigma_j^{k+1} = \sigma_j^k \sqcup f_{(i,j)}(\sigma_i^k) \tag{5.14}$$

Fixpoint computations apply the *recursive strategy* to the iterations k over F. This strategy recursively stabilizes subcomponents of every component in the w.t.o every time the component is stabilized. For every node $\{n_i, n_j, \dots\} \in N$ of depth 0, i.e. nodes that do not belong to any nested component, we know that for every edge $i \to j$, i is necessarily listed before j, i.e. $i \prec j$. Hence, the value of i used in the computation of j already has its final value, which implies that the interpretation of sequential statements is made only once and in the right order, yielding a *flow-sensitive* data-flow analysis.

In the case of a loop, the stabilization of the corresponding component is detected by the stabilization of its "head". If the value associated to the head of the component remains unchanged after a subsequent iteration, then the argument used to prove the fact that no iteration is necessary over the nodes of depth 0 shows that the values associated to the nodes within the component will not change when the equations are applied once more. Therefore, for having F^δ defined as an ultimately stationary increasing chain ($\delta < \lambda$) [34], it is sufficient to have the chain F^δ stationary for the head of the loop, for the whole loop to be stable [20].

Hence, we apply the Kleene first recursion theorem [85] to devise a constructive method of fixpoint computation: given the function space $F = \langle f_{(1,h)}, \ldots, f_{(i,j)}, \ldots, f_{(k,n)} \rangle$, where n is the number of instructions in a machine program, and the invariants vector $\langle \sigma_1, \sigma_2, \ldots, \sigma_n \rangle$, then the least fixpoint of F, $lfp\, F$, is achieved after δ iterations when, $\forall (i,j) \in E$, the following equality test holds:

$$F^{\delta}(\langle \sigma_1, \ldots, \sigma_{j-1}, \sigma_j, \ldots, \sigma_n \rangle) = \langle \sigma_1, \ldots, \sigma_{j-1}, f_{(i,j)}(\sigma_i), \ldots, \sigma_n \rangle \tag{5.15}$$

Then, starting with the bottom element in iteration 0 by defining $\Sigma[\![P]\!]^0 = \bot_\Sigma$, the $lfp^{\sqsubseteq}_{\bot_\Sigma} F = \bigsqcup_{\delta \geqslant 0} F^{\delta}$ is computed by the upward abstract iteration sequence:

$$\Sigma[\![P]\!]^{k+1} = \begin{cases} \Sigma[\![P]\!]^k & \text{if } \Sigma[\![P]\!]^k = F(\Sigma[\![P]\!]^k) \quad (5.16) \\ \sigma_j^{k+1} = \sigma_j^k \sqcup f_{(i,j)}(\sigma_i^k) & \forall (i,j) \in E : |\, w(i) \,| > 0 \quad (5.17) \\ \sigma_j^{k+1} = f_{(i,j)}(\sigma_i^k) & \text{otherwise} \quad (5.18) \end{cases}$$

This sequence chaotic iterations is declaratively defined in Haskell by the function *chaotic*. Required are the definition of an upper semilattice (*Lattice a*), which contain the *join* operation ($\sqcup$) used in Def. (5.14) and the *bottom* element, and the definition of an input-output transition (*Transition a*), which define the *source* and *sink* programs labels for an arbitrary syntactical sequence **Expr**.

```
class Lattice a where
  bottom :: a
  join   :: a → a → a
class Transition a where
  sink :: a → Label
  source :: a → Label
  expr :: a → Expr
```

The *chaotic* function takes three arguments. The first argument is the transition relation defined by the edge (i,j) as defined in (5.15). The second argument is the actual invariants map, of type *Invs a*, which will be updated after the chaotic iteration k. The third argument has the polymorphic type a and denotes the abstract value obtained after applying $f_{(i,j)}(\sigma_i^k)$. The definition of the data-flow function $f_(i,j)$ in the context of the two-level denotational meta-language will be given in Section 5.2.

```
chaotic :: (Eq a, Lattice a, Transition r) ⇒ r → Invs a → a → Invs a
chaotic rel invariants s = let s' = read invariants (sink rel)
                               s'' = join s' s
                           in if s' ≡ bottom
                                then store rel s invariants
                                else if s'' ≢ s'
                                     then store rel s'' invariants
                                     else stabilize rel invariants
```

The abstract value computed in the previous k iteration is instantiated by the variable s' and is obtained by reading from the *invariants* map the value associated with the *sink* label of the input transition relation, *rel*. If this value is equal to *bottom*, which correspond to the case defined in (5.18), then the function *store* is used to insert the value s, which corresponds to iteration $k+1$ and is taken as an input argument, directly into the *invariants* map.

Otherwise, the fixpoint condition of Def. (5.16) is tested in order to determine if the fixpoint has been reached at the program point j, or if it is necessary to proceed to the next iteration, using the value σ_j^{k+1} as input to the next data-propagation function. When the case of Def. (5.16) holds, the function *stabilize* flags the fixpoint condition as **True** for the value associated with the *sink* label of the input-output relation.

```
data Env a = Env {value :: a, stable :: Bool}

stabilize :: Transition r ⇒ r → Invs a → Invs a
stabilize rel = if (head ∘ source) rel
                then adjust (λn → n {stable = True}) $ (point ∘ sink) rel
                else id
```

For this purpose, a new record function called *stable* is added to the datatype (**Env** a). As already mentioned, loop stabilization can be detected at the head of the loop. Therefore, the flag *stable* is updated only if the *sink* label matches a **Head** label. In order to find the program point of a **Label** and to determine the type of label, is defined an instance of the type class (*Labeled* a). The function *point* returns the *labelId* of a label **Identifier** and the function *head* detects if a label has the constructor **Head**.

```
class Labeled a where
  point :: a → Lab
  head :: a → Bool
```

Finally, in those cases where Def. (5.17) holds, the *join* between the value computed during the previous iteration and the value taken as argument is computed and stored inside the returned invariants map.

The read and write interactions with the invariants map is defined by means of the type class (*Container* a b), using the functions *read* and *store*. The type variables a and b stand for the container type (*Invs* a) and the contained type (a), respectively. The function *read* retrieves from the *invariants* map the *value* stored at the program *point* of the input *label*. The function *store* updates the input value s at the program *point* of the *sink* label of the input relation *rel*.

```
class (Lattice b) ⇒ Container a b where
  read :: (Labeled l) ⇒ a → l → b
  store :: (Transition r) ⇒ r → b → a → a

instance (Lattice a) ⇒ Container (Invs a) a where
  read invariants label = value $ invariants ! (point label)
  store rel s = adjust (λn → n {value = s}) $ (point ∘ sink) rel
```

5.2 Meta-Language

This section delineates Contrib. (i) as the definition of a polymorphic, two-level meta-language [100], capable to express the semantics of different programming languages in a unified fixpoint semantics. The same meta-program can be parameterized by different state-propagation semantic functions, defined for a variety of abstract domains. Automatic generation of type-safe fixpoint interpreters is obtained directly in Haskell by providing interpretations to the meta-language combinators using λ-calculus. This is a step forward to carry out Contrib. (ii).

The hierarchy of semantics of a transition system by abstract interpretation proposed in [27], in particular the existing isomorphism between the relational and the denotational program semantics, is used to design our meta-language based on algebraic relations. In this way, we achieve Contrib. (iii) by letting the semantic functions defined in Section 5.1 to be used as relations in the context of the two-level denotational meta-language in order to compute fixpoints using a reflexive transitive closure.

An important method for defining a data-flow analysis framework is to give an interpreter for the the language that is written in a second language with better characteristics from the analysis point of view, e.g. the adequacy for computations with symbolic expressions. In [110], these two languages are designated by *defined* and *defining* ("host") languages, respectively. For example, a *defined* language with *imperative* features, such as statement sequencing, labels, jumps, assignment, and procedural side-effects, can be interpreted by a *defining* language with *applicative* features, such as the evaluation of expressions and the definition and application of functions.

Definitional interpreters must include expressions of the defined language. In order to avoid questions related to grammars and parsing, the approach of *abstract syntax* was introduced in [87]. In this approach, syntactical phrases of programs are represented by abstract, hierarchically structured data objects. The concept of data object is used by Reynolds in [110] to introduce the process of *defunctionalization* in the formalization of a definitional interpreter where any of its functions cannot accept arguments or produce results that are functions. Operationally, defunctionalization removes function types from the type signatures of the defining interpreter and replaces them with a coproduct datatype. The original effects are obtained by introducing an "apply" function that interprets the coproduct and returns the expected functional type.

When accompanied with the techniques of *closure conversion* and transformation into *continuation-passing style* (CPS), defunctionalization define the structure of an abstract machine for the lambda-calculus [4]. Concretely, a closure-converted interpreter has first-order data flow and a CPS-transformed interpreter has sequentialized control flow. Moreover, this kind of defining interpreters can be mechanically obtained as the counterpart of the corresponding

"compositional" interpreters. Suppose, however, that higher-order functions are convenient for defining interpreters using, for example, a denotational semantics to perform data flow analysis [94]. In these cases, a technique called *refunctionalization* [41] can be used to refunctionalize an abstract machine into a higher-order counterpart.

Refunctionalization proceeds by reversing the steps of defunctionalization. Given a first-order datatype δ and an "apply" function of type $\delta \times \tau \rightarrow \tau'$ dispatching the datatype in the image of Reynold's defunctionalization algorithm, the "apply" function can be refunctionalized into the functional type $\tau \rightarrow \tau'$. Moreover, in [41] is stated that refunctionalized abstract machines always give rise to continuation-passing programs.

In the presence of complex control flow, e.g. involving labels and jumps, the benefits of having higher-order and compositional defining interpreters are of great importance when the data flow analysis framework is based on a denotational semantics. On the other hand, defunctionalized interpreters can by applied to generate code for various abstract machines. These two features are indirectly related to the two-level denotational meta-language proposed in [99, 100]. When using abstract interpretation to specify data flow analysis and proving it correct, the distinction between a denotational "macro-semantics", which is defined at compile-time, and a denotational "micro-semantics", which is defined for run-time domains, paves the way for a systematic treatment of data flow analysis [70, 94], to perform program transformation [22] and for automatically compiler generation [77, 126]. More recent approaches, for example [5], comprise data/control flow unification by taking advantage of the algebraic properties of higher-order control-flow in order to compose first-order data-flow computations.

The denotational framework for data flow analysis proposed in [94] establishes that a continuation style formulation of denotational semantics naturally leads to the MOP (*merge over all paths*) fixpoint solution, whereas a direct style formulation of denotational semantics leads to the MFP (*minimal fixed-point*) fixpoint solution. In the earliest approaches to data flow analysis [72], program semantics were considered from an operational point of view that is not syntax-directed. On the contrary, the denotational approach is based on "homomorphisms" from syntax to denotations. One advantage of this approach is the establishment of a clear connection between the MFP and MOP fixpoint solutions. In particular, the MOP solution can always be specified as the MFP solution to a different data flow analysis problem.

5.2.1 Declarative Approach

The previous section introduced the notion of *meta-trace semantics* of a program and explained the process of computing fixpoints using this semantic characterization by means of data-propagation functions that express the data dependencies on the program. Therefore, in order to effectively compute the resulting Kleenian least fixpoints, one needs to

instantiate the propagation functions that, in some sense, abstract away from the history of computations. In fact, data-propagation denotational functions are "local", side-effect free, input-output functions. In this way, the ideal semantic formalism to express the propagation functions would be a nondeterministic denotational semantics [27].

The arising challenge is then to find a sound mechanism that correlates all the semantics formalisms in question in a consistent way. In [27], Cousot presents a constructive design of an hierarchy of semantics by abstract interpretation in which sound approximations between the fixpoint semantics computed at trace, relational and denotational levels are defined. In fact, each level of abstract has its practical advantages. At trace-level, we find a concatenation of program states which order is specified by a weak topological order and an iteration strategy. At relational-level, we find a convenient way to label the compiled machine program and to change the size of the basic blocks by increasing/decreasing the length of the syntactical objects inside each relation. At denotational-level, we have a natural way to perform functional applications, to compute effects and to apply fixpoint mathematical definitions [74, 133].

To put in practice this purpose of correlation of semantic formalisms, we employ a modified version of the two-level denotational meta-language introduced by Nielson [100]. Put simply, our objective aims to express the semantics of different programming languages in a unified fixpoint form by means of algebraic relations. At the higher level of the meta-language are defined *meta-programs* that encode the control flow graph of the program, which interpretation is always the same, regardless of the abstract domain. Meta-programs depend only on compile-time information, more precisely on their weak topological orders. Moreover, the semantics of meta-programs exhibits good properties for program transformation due to its relational-algebraic shape [16, 121].

At the lower level of the meta-language are defined different abstract interpretations, in the form of denotational semantic functions, that are used to compute transformations on abstract program properties. In complement to the higher level, these semantic state transformers express the particularities of some programming language and are used to parameterize the relational algebra at the upper level [115]. Pragmatically, this separation in two levels brings the possibility to: first derive a *meta*-program, composed by relational operators, which reflects the structure of the program; and second, *simulate* the meta-program in the abstract domain, by means of chaotic iteration strategy, so that the fixpoint computations mimic program executions, passing the denotational state-propagation functions as arguments to the combinators defined at the higher-level of the meta-language.

More specifically, the two levels of the meta-language distinguish between high-level *compile-time* (ct) entities and low-level *run-time* (rt) entities. At the higher-level, *meta*-programs are compositionally expressed in relational terms by means of binary relational combinators. The advantage of this approach is that new programs can be obtained throughout the composition

of smaller programs, in analogy to graph-based languages. Implemented combinators are the sequential composition ($*$), the pseudo-parallel composition ($||$), the intra-procedural recursive composition ($\oplus$), and the inter-procedural recursive composition ($\oslash$). At the lower level, semantic transformers of type $\mathrm{rt}_1 \rightarrow \mathrm{rt}_2$ provide the desired denotational effects.

$$\mathrm{ct} \triangleq \mathbb{B} \mid \mathrm{ct}_1 * \mathrm{ct}_2 \mid \mathrm{ct}_1 \mid\mid \mathrm{ct}_2 \mid \mathrm{ct}_1 \oplus \mathrm{ct}_2 \mid \mathrm{ct}_1 \oslash \mathrm{ct}_2 \mid \mathrm{split} \mid \mathrm{merge} \mid \mathrm{rt} \tag{5.19}$$

$$\mathrm{rt} \triangleq \Sigma \mid (\Sigma \times \Sigma) \mid \mathrm{rt}_1 \rightarrow \mathrm{rt}_2 \tag{5.20}$$

The two-level meta-language unifies data and control flow in a pure functional language. Control flow is expressed at relational level by the upper level of the meta-language using the *point-free* notation [52]. Using this notation, the input state to a composition of propagation functions, either ($*$), ($||$), ($\oplus$) or ($\oslash$), is associated with left parenthesis with the output state. In this way, references to the input argument can be removed from the compositional layer (*point-free*), allowing the compositional combinators to become binary relational operators by taking two relations as arguments and producing a new relation. Note that there are no bound variables at the higher-level of the meta-language. For each combinator, there is only one bound variable that corresponds either to an input state with the type Σ or the product $(\Sigma \times \Sigma)$. We let $\mathbb{B}$ denote the logical boolean values (*True* and *False*), such that $\mathbb{B} \triangleq \{\mathrm{tt}, \mathrm{ff}\}$.

Data flow is defined at the lower level of the meta-language by means of state-propagation functions, which are extensions to program states of the data-propagation functions of Def. (5.11). Next, we will detail how instances of these state-propagation functions are obtained as abstractions of the relational semantics. As mentioned in Section 5.1, transition relations $\tau \subseteq (\Sigma[\![P]\!] \times \mathrm{Instrs}[\![P]\!] \times \Sigma[\![P]\!])$ are ternary relations which, given a syntactic object $i \in \mathrm{Instrs}[\![P]\!]$, establishes a input-output relation between a state and its possible successors in the context of some program P. Assuming the abstract state vector $\Sigma[\![P]\!] = \langle \sigma_1, \sigma_h, \ldots, \sigma_i, \sigma_j, \ldots, \sigma_n \rangle$, where n is a label identifier, we then apply the relational abstraction defined in [27], using the right-image isomorphism f applied to every transition relation τ such that:

$$\begin{aligned} f_{(i,j)}[\![\tau]\!] \triangleq \lambda\sigma_i \bullet (\sigma_j \mid & \exists \Sigma[\![P]\!]^i, \Sigma[\![P]\!]^j : \sigma_i = \Sigma_P^i[i] \wedge \sigma_j = \Sigma_P^j[j], \\ & \exists \iota \in \mathrm{Instrs}[\![P]\!] : \langle \Sigma_P^i, \iota, \Sigma_P^j \rangle \in \tau) \end{aligned} \tag{5.21}$$

Each function $f_{(i,j)}$ is partially applied to the syntactic object $\iota \in \mathrm{Instr}$, so that, at denotational level, we reason on functions exclusively with the type $(\mathbb{C} \rightarrow \mathbb{C})$, only by using the abstract values located at the labels i and j. When computing the MFP fixpoint solution, the least upper bound of the multiple states arriving at the program label j is computed. Afterwards, the type of the "local" data-propagation function f is lifted to the global state functional type $(\Sigma \rightarrow \Sigma)$ in order to obtain the space of state-propagation functions $F = \langle f_{(1,h)}, \ldots, f_{(i,j)}, \ldots, f_{(k,n)} \rangle$, where each $f_{(i,j)}$ is defined by (5.21) and n is the number of instructions in the machine program. Any functional types are straightforwardly defined

in Haskell by the parametric polymorphic type ($RelAbs\ a$) and the relational abstraction is defined by the type class ($Abstractable\ a$).

```
type RelAbs a = a → a
```

The function *apply* receives as argument one instance of an invariants semantic transformer with type ($RelAbs\ (Invs\ a)$), and returns a state transformer, which type ($RelAbs\ (\mathbf{St}\ a)$) denotes the run-time type ($\Sigma \to \Sigma$). The instance of the invariants semantic transformer is produced by the function *lift*, which provided with a transition relation of type ($\mathbf{Rel}\ a$) and a propagation function of type ($RelAbs\ a$).

```
class Abstractable a where
  apply :: RelAbs (Invs a) → RelAbs (St a)
  lift :: Rel a → RelAbs a → RelAbs (Invs a)
```

The right-image isomorphism of Def. (5.21) is defined in Haskell by the function *refunct*, which takes as argument one relation parametrized by a state, $\mathbf{Rel}\ (\mathbf{St}\ a)$, and returns a relational abstraction ($RelAbs\ (\mathbf{St}\ a)$). The first *step* is to instantiate the abstract semantic transformer *dataFlow* as an interpretation of a syntactic phrase **Expr**. The resulting function has type ($RelAbs\ a$) and is then lifted to the domain of program invariants ($Invs\ a$) using the function *lift*. Finally, the semantic function between two program states has type ($RelAbs\ (\mathbf{St}\ a)$) and is obtained using the function *apply*.

```
dataFlow :: Expr → a → a

refunct :: (Abstractable a, Transition (Rel (St a))) ⇒ Rel (St a) → RelAbs (St a)
refunct r = let step = dataFlow (expr r)
            in apply $ lift r step
```

A generic instance of the ($Abstractable\ a$) type class is given next. The function *lift* takes as arguments the relation r, and the relational abstraction f, which is defined for the polymorphic type a (abstract domain). The lifted version reads the existing abstract value from the invariants map at the *source* label of r, applies f to it, and then passes the result to the function *chaotic* previously defined in Section 5.1.

```
instance (Lattice a, Transition (Rel (St a)), Eq a) ⇒ Abstractable a where
  apply f s@St {invs = i} = s {invs = f i}
  lift r f i = let s = read i (source r)
               in chaotic r i (f s)
```

In practice, using the semantic projection mechanism defined in [27], the fixpoint algorithm evaluates meta-programs at trace level, by expanding the meta-trace according to iteration strategy, but using the program's structural constructs defined at relational level and the program's functional behavior defined at denotational level. Hence, the fixpoints semantics $lfp^{\sqsubseteq}_{\perp_\Sigma} F$ is defined in terms of data-propagation functions $f_{(i,j)}$, where i and j are such that $\langle \Sigma^i_P, \iota, \Sigma^j_P \rangle \in \tau$ and ι denotes a syntactic expression of P. Assuming that multiple incoming nodes n_k arriving to the node n_i may exist, but that only one outgoing node n_j leaves from n_i, the fixpoint denotational semantics is defined as abstraction of the relational semantics:

$$F(f_{(i,j)}) \triangleq \lambda f_{(i,j)} \bullet \lambda \Sigma_P^i \bullet (\Sigma_P^j[j] := f_{(i,j)}(\sigma_i) \mid \forall k : \sigma_k = \Sigma_P^k[k], \exists i : \sigma_i = \bigsqcup f_{(k,i)}(\sigma_k) : \Sigma_P^k \, \tau \, \Sigma_P^i \wedge \Sigma_P^i \, \tau \, \Sigma_P^j) \tag{5.22}$$

If fact, the previous definition provides an approximation to the MOP fixed-point solution by providing a constructive iteration method based in functional application and least upper bounds operators. Compared to the natural nondeterministic fixpoint semantics given in [27], two observations should be made: the first is that Def. (5.22) gives an abstract (approximate) fixpoint semantics of a nondeterministic system of input-output relations, whereas in [27, Theorem 33] gives the *collecting semantics* of such system; the second is that both definitions must be different because the state transformer used in the collection semantics is *additive*, i.e. a complete join morphism, where the abstract state transformer used in Def. (5.22) is not additive, but only continuous. Consequently, according to [35, Section 9], the MFP solution computed by Def. (5.22) is an over-approximation of the MOP solution, which definition was provided by Def. (5.12) in Section 5.1.

Example 4. Derivation of a meta-program using higher-order combinators. Fig. 5.4 illustrates how a meta-program is derived for the machine code example in Fig. 5.2 of Example 2 and the corresponding iteration strategy (5.13) of Example 3. Starting with the first input-output relation defined for a syntactical expression, the first state-propagation function is instantiated for the edge given by the pair of labels identifiers (0,1), having the instruction '`mov ip, sp`' as the partially applied syntactic object. Then, the meta-program is constructed as an interpretation of the weak topological order, during which state-propagation functions are instantiated along the program paths and composed using the control flow combinators defined in the upper level of the meta-language.

```
(mov ip, sp) * ··· * (mov r0, #5) * (bl 24) * ··· * (str r0, [fp, #-16]) * (b 16) *
··· * (ldr r3, [fp, #-16]) * (cmp r3, #0) * ((bgt -20) ⊕ (ldr r3, [fp, #-16]) * ··· *
(sub r3, r3, #1) * ··· * (ldr r3, [fp, #-16]) * (cmp r3, #0)) * (bgt -20) * ··· *
(mov r0, r3) * (ldmfd sp, r3,fp,sp,pc) * (mov r3, r0) * (str r3, [fp, #-16]) * ··· *
(ldmfd sp, r3,fp,sp,pc)
```

Figure 5.4: Fragments of a meta-program derived from an iteration strategy and a relational semantics

For the '**while**' loop contained in the source program in Fig. 5.1, and by inspection of the iteration strategy (3), a feedback edge is detected between the label indentifiers 22 and 17, such that $22 \preceq 17 \wedge 22 \in w(17)$, according to Def. (5.9). This explicitly states that a recursive pattern was found in the control flow of the program. Accordingly, the corresponding meta-(sub)program uses the binary relational operator $(\cdot \oplus \cdot)$ to compose the state-propagation function for the *branch* instruction '`bgt -20`', which is at the head of the loop, with another

meta-subprogram containing the body of the loop, which is a sequential composition (`*`) of state-propagation functions for a list of instructions, starting with '`ldr r3, [fp, #-16]`' and ending with '`cmp r3, #0`'. ▲

Next, we give the formal definitions of the binary relational combinators as denotational interpretations according to the notion of subgraphs. Fig. 5.5 gives the graph-based representation of the meta-programs. As expected, the simpler control-flow patterns are 'sequencial', 'pseudo-parallel', 'alternative' and 'recursive' compositions. Interpretations for the interface adapters 'split' and 'merge' are also given.

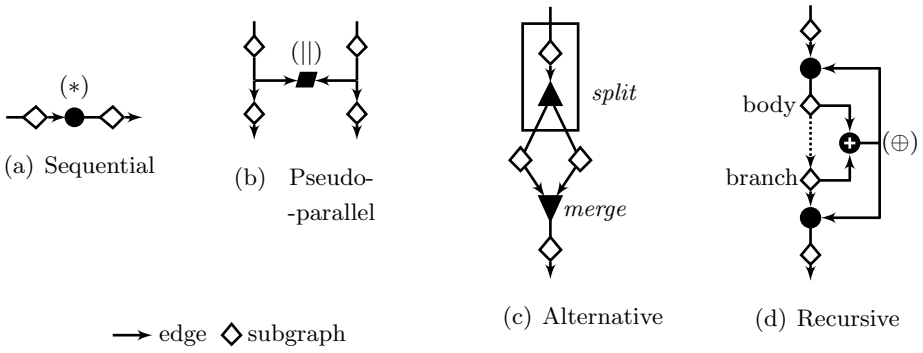


Figure 5.5: Graph-based representation of the control-flow patterns

Every time two edges are connected by consecutive labels, we apply the sequential composition of the two corresponding subgraphs. The sequential composition $(\cdot * \cdot)$ of two relations T and R is defined by $a(T * R)c$ iff there exists b such that aTb and bRc. In point-free notation, its type is $T * R :: a \to c$. Since functions are considered to be a special sort of relations, we give the following Haskell definition for the combinator $(\cdot * \cdot)$:

$$(*) :: (a \to b) \to (b \to c) \to (a \to c)$$
$$f * g = \lambda s \to (g \circ f)\ s$$

When two edges have two different source labels and two different target labels, we apply the pseudo-parallel composition of the subsequent subgraphs. The pseudo-parallel composition $(\cdot \,||\, \cdot)$ of two relations T and R is defined by $(a, c)(T \,||\, R)(b, d)$ iff $aTb \wedge cRd$. Its point-free type is $(a, c) \to (b, d)$ and the Haskell definition is:

$$(\,/\,) :: (a \to b) \to (c \to d) \to ((a, c) \to (b, d))$$
$$f \,/\, g = \lambda(s, t) \to (f\ s, g\ t)$$

The interpretation of a intra-procedural loop corresponds to the reflexive transitive closure of the subgraphs that constitute the loop. Hence, the loop structure is divided between body of the loop T and the branch condition R. The recursive composition $(\cdot \oplus \cdot)$ of two relations T and R is defined by $a\ (T \oplus R)\ a$, where a is a type variable. The conjunction $(b\ T\ a) \wedge (a\ R\ b)$ corresponds to loop unrolling, i.e. when the recursive operator $(\cdot \oplus \cdot)$ invokes itself in a tail-recursive manner until the fixpoint condition is satisfied; otherwise the output when the

fixpoint of the loop was achieved has the same type a of the free variable. Therefore, in point-free notation, its type is $T \oplus R : a \to a$.

The corresponding Haskell definition is $(+)$. The free variable s of type a is used to compute the value at the output of the branch condition, which is produced by the argument function r, in order to detect the fixpoint condition provided by the function *isFixpoint*.

```
( + ) :: (Iterable b) ⇒ (b → a) → (a → b) → (a → a)
t + r = fix $
        λrec s → let s' = r s
                 in if ¬ $ isFixpoint s'
                    then (t ∗ rec) s'
                    else compl s
```

The function *fix* uses the typed lambda calculus of Haskell to allow the definition of recursive functions. Let a be a polymorphic type variable. The definition of *fix* is $\lambda f \to f(\mathit{fix}\ f)$ and its type $(a \to a) \to a$ derives directly from the Kleene's first recursion theorem. Intuitively, the application *fix* f yields an infinite application stream of fs: $f(f(f(...)))$. However, this sequence is only a Kleene sequence if f is continuous and if the sucessive application of f form an ascending chain [74]. Hence, it is necessary to provide a function that is able to specify the boolean condition $\mathbb{B}$ of Def. (5.19), which determines the end of the recursive (iterative) applications of f.

This mechanism is provided by the type class (*Iterable* a), in particular by the function *isFixpoint*. The second function defined in the type class is *emptyStack* and is used to check wether the state of the procedure call stack is empty or not and is necessary for a *context-sensitive* interpretation of interprocedural loops, as will be described latter in Section 6.5.

```
class Iterable a where
  isFixpoint  :: a → Bool
  emptyStack :: a → Bool
```

Finally, a *path-sensitive* analysis of loops requires the definition of the function *compl*, which is used to update a specific flag in the input state value s, stating that the loop test condition turned to 'ff' after the fixpoint has been reached for the recursive subgraph. This will enforce the analysis of the fall-through loop condition. The function *compl* is also used by the interface adapter *split* before invoking the Haskell pseudo-parallel combinator $(/)$, in which two alternative, i.e complementary in terms of some given conditional expression, program paths are represented.

```
compl :: a → a
```

The interpretation of a inter-procedural recursive subgraph corresponds to the reflexive transitive closure of the subgraph that constitutes the procedure. Similarly to the intra-procedural loop, let T denote the body of the loop and R denote the procedure "return" relation. Then, the inter-procedural recursive composition $(\cdot \oslash \cdot)$ of two relations T and R is

defined by $a\ (T \oslash R)\ b$, where a and b are types variables. The two differences in the definition of this operator when compared to the definition of the intra-procedural loop operator $(\cdot \oplus \cdot)$ are the use of the function *emptyStack* instead of *isFixpoint* and the fact that the output state value does not require any path instrumentation. In fact, the function *emptyStack* simply returns 'tt' while the recursive procedure call stack is empty, e.g. after completing the analysis of multiple recursive calls to a procedure.

$$
\begin{aligned}
&(\ \%\ ) :: (\mathit{Iterable}\ c) \Rightarrow (b \to a) \to (a \to b) \to a \to b \\
&t\ \%\ r = \mathit{fix}\ \$ \\
&\qquad \lambda \mathit{rec}\ s \to \mathbf{let}\ s' = r\ s \\
&\qquad\qquad \mathbf{in\ if}\ \neg\ \$\ \mathit{emptyStack}\ s' \\
&\qquad\qquad\qquad \mathbf{then}\ (t * \mathit{rec})\ s' \\
&\qquad\qquad\qquad \mathbf{else}\ s'
\end{aligned}
$$

As already mentioned, interface adaptation is required prior and after pseudo-parallel composition. In the former case, the *split* function has type $(a \to (a, a))$ and returns a pair of values consisting in the previous subgraph output, captured by the bound variable s of type a, together with the corresponding "complemented " value that is required by the path-sensitive analysis.

$$
\begin{aligned}
&\mathit{split} :: a \to (a, a) \\
&\mathit{split} = \lambda s \to (s, \mathit{compl}\ s)
\end{aligned}
$$

Afterwards, the pseudo-parallel combinator $(/)$ can be applied to alternative program paths. In the latter case, when the output of two pseudo-parallel subgraphs need to be combined into a single state, the function *merge* is applied. It may happen that, during path-sensitive analysis, one of the alternative paths is infeasible while the other path is feasible. In these cases, it is necessary to instrument the values at the end of the infeasible paths so that the joined state becomes feasible again. To this end, the type class $(\mathit{Infeasible}\ a)$ is defined. In any case, the output value is computed using the *join* function of the $(\mathit{Lattice}\ a)$ type class.

$$
\begin{aligned}
&\mathit{merge} :: (\mathit{Lattice}\ a, \mathit{Infeasible}\ a) \Rightarrow (a, a) \to a \\
&\mathit{merge} = \lambda(a, b) \to \mathbf{case}\ (\mathit{isNotFeasible}\ a, \mathit{isNotFeasible}\ b)\ \mathbf{of} \\
&\qquad\qquad\qquad\qquad (\mathbf{False}, \mathbf{False}) \to \mathit{join}\ a\ b \\
&\qquad\qquad\qquad\qquad (\mathbf{False}, \mathbf{True}) \to \mathit{join}\ a\ (\mathit{becomeFeasible}\ b) \\
&\qquad\qquad\qquad\qquad (\mathbf{True}, \mathbf{False}) \to \mathit{join}\ (\mathit{becomeFeasible}\ a)\ b \\
&\qquad\qquad\qquad\qquad (\mathbf{True}, \mathbf{True}) \to \mathit{join}\ a\ b \\
&\mathbf{class}\ \mathit{Infeasible}\ a\ \mathbf{where} \\
&\quad \mathit{isNotFeasible} :: a \to \mathit{Bool} \\
&\quad \mathit{becomeFeasible} :: a \to a
\end{aligned}
$$

Let $b ::= (\cdot * \cdot) \mid (\cdot \ ||\ \cdot) \mid (\cdot \oplus \cdot) \mid (\cdot \oslash \cdot)$ be the syntactical meta-variable for the binary combinators in the upper level of the meta-language. Let also the interface adapters 'split' and 'merge' be represented by input-ouput relations. Then, the reflexive transitive closure $T^\star$ of the program's initial input-output relation T, where R is a bound input-output relationis expressed in point-free style as a generalization on the Kleene/Knaster/Tarski fixpoint theorem [28]:

$$T^{\star} \triangleq \bigsqcup_{n \geqslant 0} T^{n} = \bigsqcup_{\substack{n \geqslant 0 \\ i \leqslant n}} (\lambda R \bullet (T \; b \; R))^{i}(\bot_{\Sigma}) \tag{5.23}$$

where $\bot_{\Sigma}$ is the undefined abstract state. In this way, fixpoint semantics can be efficiently computed by using program-specific *chaotic iteration strategies* [20, 36], specified at compile-time level by the type expressions in the meta-language for free. In complement to type checking, the soundness of the abstract state-state transformers, which have the unified type $\mathrm{rt}_1 \rightarrow \mathrm{rt}_2$ and are defined at run-time level, are proven correct by using the calculational approach proposed in [28], as will be described latter in Sections 6.6 and 6.7.

Comparatively with MOP solution of Def. (5.12) and with the denotational fixpoint semantics defined in terms of the function of Def. (5.22), the MFP fixpoint semantics of Def. (5.23) is a redefinition of the later, where explicit references to program states are removed from the definition, using the point-free notation. Therefore, it relies only on the types of upper level of the meta-language. In this point-free relational view, fixpoints have a type safe definition for free which is suitable for algebraic transformations on the structure of programs. Examples can be the unrolling of the first loop iteration outside the recursive block or the transformation of programs with loops into purely sequential programs. The last transformation is of particular interest when performing fixpoint verification, as will be described latter in Section 7.1.

The interplay between the semantics projection mechanism and the algebraic properties of the meta-language gives origin to a *meta-semantic formalism*, which will be used to express fixpoints in different domains of interpretation in the analysis of several components of the WCET analyzer, to express contract specifications at source code level to verify the results of fixpoints computed at machine code level, and to transform programs into other programs for which the verification of the existence of fixpoints can be more efficiently done. Fig. 5.6 illustrates the way the meta-semantic formalism [27] combines the semantic projection mechanism with a transformation algebra [22] to specify and verify abstract properties of programs [29, 30] written in different programming languages [113].

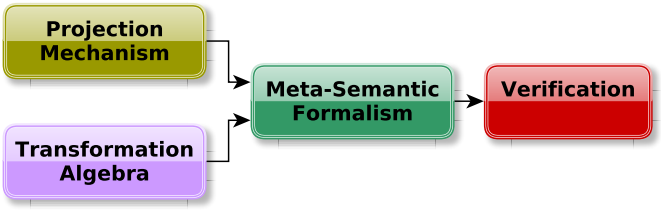


Figure 5.6: Different Interactions of the Meta-Semantic Formalism

5.3 Intermediate Graph Language

This section describes our intermediate graph language, used to encode the abstract syntax of the weak topological order of a particular program. The definition of the program's syntactic structure by means of the intermediate language improves the mechanism of the automatic generation of abstract fixpoint interpreters using a denotational semantics. In this way, Contrib. (ii) is completely specified.

The data flow analysis framework presented so far has its foundations on a theory of fixpoints, which generically considers a system of data flow equations, commonly referred as semantic functions in the context of denotational semantics. As already described in the previous Sections 5.1 and 5.2, this method for data flow analysis in the denotational setting requires that the intensional information contained in the weak topological order of the program is not lost when abstracting input-output relations into the extensional framework of the two-level denotational meta-language. By this reason, the data-flow equations of Def. (5.11) are instantiated as state-transformation functions by refunctionalizing [41] the set of input-output relations, as formally described by the right-image isomorphism of Def. (5.21) and declaratively defined in Haskell using the function *refunct*.

However, the fixpoint algorithm defined of Def. (5.23) is *compositional*, in the sense the higher-order semantic functions are combined to model control flow, but in a way such that the order of application specified by the chaotic iteration strategy is preserved and correct in the light of the type system of Haskell. Experience showed us that the abstraction effect produced by the function *refunct* cannot be performed until all the intensional information contained in the input-output relations is no longer necessary. Hence, we have defined an abstract syntax to represent control flow as a "homomorphism" between syntactic phrases using this abstract syntax and the expressions defined at the upper level of the two-level denotational meta-language.

The objective is to use the defining programming language Haskell to compile (or interpret) another definition written in Haskell to give an interpreter for the defined language. In this way, the reflexive transitive closure of Def. (5.23), can be automatically compiled as an interpretation of the abstract syntax into expressions of the upper level of the meta-language. Since each of the binary relational operators are interpreted into the lambda-calculus, the overall (compositional) effect is obtained by a Haskell program. Moreover, the semantic functions provided at the lower level of the meta-language are polymorphic in the domain of interpretation.

Our approach is based on the compilation of denotational interpreters proposed in [13]. The objective is to automatically obtain the derivation of meta-programs that specify a particular fixpoint algorithm in the form of (5.23). Let D be the language of denotational semantics and δ is a particular definition of some programming language ($\delta \in D$). Given an input

program π of type P and some input σ of type Σ, the interpretation of δ and π on σ is defined by the interpreter $C_1 \in D \times P \times \Sigma \to \Sigma$.

Further, a second interpreter $C_2[\![\delta]\!]$, of type $C_2 \in D \to P \times \Sigma \to \Sigma$, is defined by partially applying the definition δ to finally run π on σ. Again by partial application, the program $\pi \in P$ is compiled to give an interpreter $((C_3[\![\delta]\!])[\![\pi]\!])$ of type $C_3 \in D \to P \to \Sigma \to \Sigma$, which finally compiles π, as desired. The definition of C_3 is given in two steps: (1) by means of the compiler C_4, δ can be treated as a method of translating a program P into lambda-calculus Λ; (2) a compiler L for λ-calculus expressions.

$$C_4 \in D \to P \to \Lambda \tag{5.24}$$

$$L \in \Lambda \to \Sigma \to \Sigma \tag{5.25}$$

$$C_3[\![\delta]\!] \equiv L \circ (C_4[\![\delta]\!]) \in P \to \Sigma \to \Sigma \tag{5.26}$$

5.3.1 Declarative Approach

The derivation of meta-programs directly as a result from an interpretation of the weak topological order of a program P has the follow inconvenient: the program labels contained in the delimiting states of input-output relations are no longer explicitly available after performing the relational abstraction (5.21). In fact, the intensional information stored in two adjacent labels in the weak topological order is abstracted into the extensional, side-effect free, format of denotational semantics, where the datatype holding the syntactic object of a relation is eliminated by refunctionalization.

However, when in presence of alternative paths in the machine program, e.g. resulting from a block '**if-then-else**' in the source code, this becomes a technical limitation because the interpretation of the weak topological order needs "split" one path into two alternative paths after one branch instruction and afterwards "merge" the program states at the end of these paths, at some given "join" label. The technical problem arises because the comparison of the two calculated meta-programs is necessary in order to determine this particular "join" label. Nonetheless, by compositionality of the data-flow analyzer, the calculation process can continue from that label onward.

The solution found was to devise an inductive *intermediate graph language* that mimics the execution order of trace semantics and "connects" the relations τ denoted in Haskell by (**Rel** a), in order to obtain a dependency graph (**G** a). Such dependency graph encodes the control flow of the program using a abstract-syntax datatype, which contains the intensional information necessary to inspect the precise location of a label inside the dependency graph. The inductive abstract syntax of a dependency graph allows the representation of all the program paths allowed in any program, which are identified at compile-time by the weak topological order of the program.

$$\begin{aligned}\textbf{data}\ \mathbf{G}\ a = {} & \textbf{Empty} \mid \textbf{Leaf}\ (\textbf{Rel}\ a) \mid \textbf{Seq}\ (\mathbf{G}\ a)\ (\mathbf{G}\ a) \mid \textbf{Unroll}\ (\mathbf{G}\ a)\ (\mathbf{G}\ a) \\ & \mid \textbf{Unfold}\ (\mathbf{G}\ a)\ (\mathbf{G}\ a) \mid \textbf{Choice}\ (\textbf{Rel}\ a)\ (\mathbf{G}\ a)\ (\mathbf{G}\ a)\end{aligned}$$

A dependency graph is either an empty graph (**Empty**), a subgraph consisting in a single relation (**Leaf**), two subgraphs connected in sequence (**Seq**), two intra-procedural subgraphs connected recursively (**Unroll**), two inter-procedural subgraphs connected recursively (**Unfold**), or two subgraphs connected pseudo-parallelly (**Choice**).

The advantages of the intermediate graph language are mainly three: (1) an interpretation of the abstract-syntax tree enables the automatic compilation of dependency graphs into meta-programs; (2) by induction on their abstract syntax trees, dependency graphs can be transformed according to the algebraic rules of the meta-language; (3) for visualization and bug-tracking purposes, the syntax tree can be translated into the XML format of graph visual languages such as GraphML [135].

By taking advantage of the algebraic properties of the higher-order relational combinators, meta-programs are "calculated" using the denotational approach. The syntactic phrases of a program are their dependency graphs. The denotations of each component of ($\mathbf{G}\ a$) are expressed by the combinators defined in the upper level of the two-level denotational meta-language. The main advantage of using Haskell for the calculation of fixpoint interpreters is the fact that a definition written in Haskell can be compiled (or interpreted) to give a type safe interpreter. This guarantees the type correctness of the expressions using the core semantics (5.19) parametrized by the abstract state transformers defined at run-time (5.20).

The automatic compilation of dependency graphs into meta-programs can be regarded as a system for interpreting definitions to give interpreters [13]. After introducing the intermediate graph language, we will now refer to the syntactical phrases of the abstract syntax ($\mathbf{G}\ a$) as "programs". In the denotational setting, these syntactic phrases are interpreted into the core semantics of the meta-language. Afterwards, meta-programs in the form of (5.23) are automatically compiled into λ-calculus by providing a subsequent interpretation to the core semantics for the binary operators b and the unary operators u. This last step has been described in the previous Section 5.2 in the Haskell definitions of b and u.

Compared the Def. (5.26), the compiler C_4 does not translate a program into lambda-calculus terms, but rather translates a dependency graph written in the intermediate graph language, into expressions of the core semantics of the two-level meta-language. Additionally, the compiler L is an Haskell interpreter that performs the interpretations of the operators b and u into lambda-calculus. The goal of the compiler C_3 is to refunctionalize the defining interpreter to the type $(\Sigma \rightarrow \Sigma)$ when provided with a dependency graph.

The compiler C_3 is defined in Haskell by the function *derive*. By the fact the structure of dependency graphs is inductive, the type signature of *derive* requires the definition of the "continuation" meta-program f. The function *refunct* provides the right-image isomorphism used to abstract the relational semantics to the denotation level. For sake of clarify, the

refunctionalization previously defined by the function *refunct* uses the data type (**Rel** (**St** a)). Alternatively, the "higher-order" refunctionalization provided by the function *derive* uses the inductive datatype (**G** a), where the previously mentioned data type is accessible through the constructor **Leaf**.

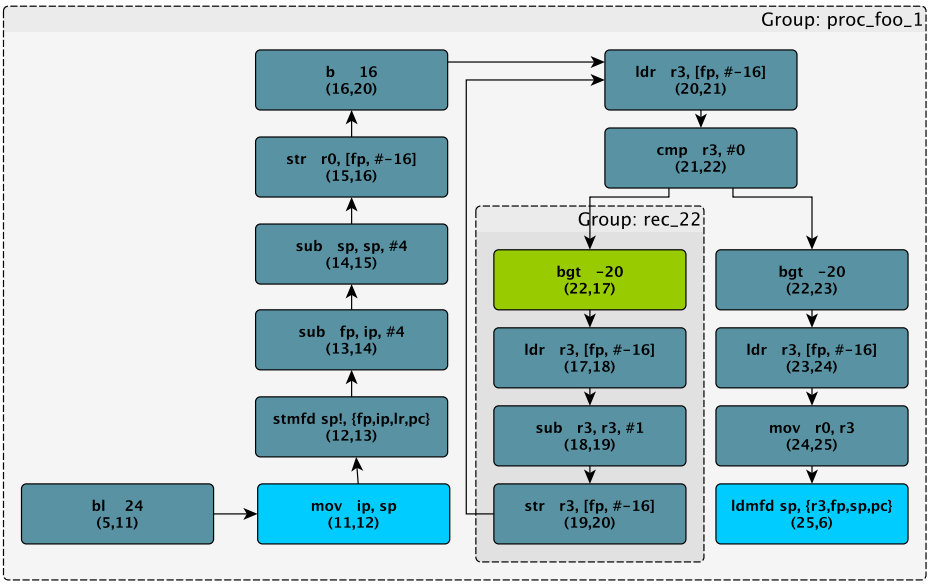
$$
\begin{aligned}
&\mathit{derive} :: (\mathit{Infeasible}\ (\mathbf{St}\ a), \mathit{Iterable}\ (\mathbf{St}\ a), \mathit{Lattice}\ a, \mathit{Lattice}\ (\mathbf{St}\ a), \mathit{Transition}\ (\mathbf{Rel}\ (\mathbf{St}\ a)), \mathit{Eq}\ a)\\
&\qquad \Rightarrow (\mathbf{St}\ a \rightarrow \mathbf{St}\ a) \rightarrow \mathbf{G}\ (\mathbf{St}\ a) \rightarrow (\mathbf{St}\ a \rightarrow \mathbf{St}\ a)\\
&\mathit{derive}\ f\ \mathbf{Empty} = \mathit{id}\\
&\mathit{derive}\ f\ (\mathbf{Leaf}\ (\mathbf{Rel}\ st)) = f * \mathit{refunct}\ (\mathbf{Rel}\ st)\\
&\mathit{derive}\ f\ (\mathbf{Seq}\ a\ b) = \mathit{derive}\ (\mathit{derive}\ f\ a)\ b\\
&\mathit{derive}\ f\ (\mathbf{Unroll}\ (\mathbf{Leaf}\ r)\ g) = f * ((\mathit{derive}\ \mathit{id}\ g) + \mathit{refunct}\ r)\\
&\mathit{derive}\ f\ (\mathbf{Unfold}\ (\mathbf{Leaf}\ r)\ g) = f * ((\mathit{derive}\ \mathit{id}\ g) + \mathit{refunct}\ r)\\
&\mathit{derive}\ f\ (\mathbf{Choice}\ a\ b\ c) = \mathbf{let}\ \mathit{left} = \mathit{derive}\ \mathit{id}\ b\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{right} = \mathit{derive}\ \mathit{id}\ c\\
&\qquad\qquad\qquad\qquad\quad\ \mathbf{in}\ f * \mathit{split} * (\mathit{refunct}\ a * \mathit{left}\ /\ \mathit{right}) * \mathit{merge}
\end{aligned}
$$

Using the function *derive*, we have demonstrated how the basic state transformers of polymorphic type (**St** $a \rightarrow$ **St** a) can be used to compile compositional fixpoint interpreters using a calculational approach. Indeed, this derivation process is polymorphic on the type a which makes the fixpoint algorithm generic in the abstract domain.

Another important advantage of the intermediate graph language and the automatic compilation of meta-programs is the possibility to generate visual representations of the dependency graph. The representation format chosen to store these graphs is the XML-based language GraphML. As a first example, consider the machine program in Fig. 5.2. For the procedure 'foo', the dependency graph induced from the weak topological order of the corresponding set of instructions is given in Fig. 5.7(a).

Example 5. Visual representation of a dependency graphs.
Although the rooted dependency graphs $G = (N, E, r)$ introduced in Section 5.1, are similar to control flow graphs, there is a fundamental difference. In dependency graphs, program execution steps are represented by the set of edges E. On the other hand, the graph notation used by GraphML specifies the dependency graph as data flow graphs. In this way, the bright blue nodes in Fig. 5.7(a) represent the first ('`mov ip, sp`') and last ('`ldmfd sp, r3,fp,sp,pc`') instructions of the procedure 'foo', respectively. The green node represents the instruction at the head of the loop ('`bgt -20`') and the nodes labelled with the pairs (20,21) and (21,22) are outside the recursive block 'rec_22' because these instructions also belong to the fall-through program path of the loop. Nevertheless, they are also included in the body of the loop, as can be inspected in the meta-program of Fig. 5.4 of Example 4.

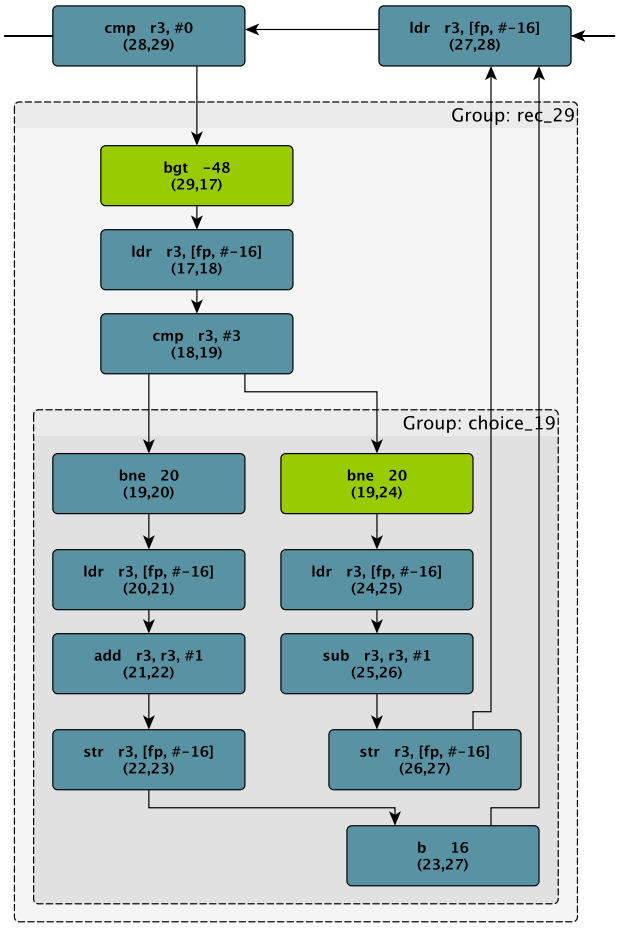
To illustrate conditional alternative paths, consider the fragment of the dependency graph in Fig. 5.7(c) for the source code in Fig. 5.7(b). A new group 'choice_19' is created to represent the '**if-then-else**' statement, where the start label is 19 and the "join" label is 27.



(a) Dependency graph of the 'foo' procedure in GraphML

```
int main(void)
{
  int y = foo2 (5);
  return y;
}

foo2 (x)
{
  while (x>0)
  {
    if (x == 3)
    {
      x++;
    }
    else
    {
      x--;
    }
  }
  return x;
}
```

(b) Source code example with an '**if-then-else**' block



(c) Dependency graph of the 'foo2' procedure in GraphML

Figure 5.7: Examples of dependency graphs visualized in GraphML

▲

5.4 Summary

This chapter describes a generic and compositional approach to data flow analysis which is suitable for the automatically generation of type safe abstract interpreters and their parameterization to compute different static analysis. The abstract properties of interest for a particular data-flow analysis are are associated to program points by a partitioned system of data-flow equations, which is solved by a chaotic iteration strategy. In practice, the dependency graph of the system of equations must be ordered according to a weak topological order to perform a flow-sensitive analysis. Using an abstract syntax to represent the control flows between program labels, an interpreter recursively traverses the dependency graph and automatically compiles a higher-order, continuation-passing and compositional fixpoint algorithm.

Since program states are labelled according to a weak topological order, and there are only a finite number of program labels, we introduce the concept of meta-trace as the cornerstone of the calculational design of reflexive transitive closures. Finally, given the definition of an abstract domain, suitable for a particular data flow analysis by abstract interpretation, the evaluation of a meta-trace results in the MFP solution of the system of data-flow equations, although this solution is computed using a MOP-style, i.e. compositional algorithm. In fact, the MOP solution can always be specified as the MFP solution to a different data flow analysis problem [94], whose MFP solution need not be computable [35].

Chapter 6

WCET Analyzer

This chapter presents a detailed description of all the components that constitute the WCET analyzer. As stated in Contrib. (iv), we instantiate the generic data flow framework described in Chapter 5, to perform a single data flow analysis, i.e. an analysis based on a single semantic transformer defined at the lower level of the meta-language (5.20), that simultaneously computes the *value analysis*, the *cache analysis* and the *pipeline analysis* of the machine program running on an ARM9 microprocessor[1]. In practice, the *pipeline analysis* uses the intermediate states of the other two static analysis with the objective to compute an abstract pipeline semantics capable of providing a set of execution times, local to each program point.

According to Contrib. (v), the abstract state transformers defined for the *value analysis* and *cache analysis* are proven to be "correct by construction", in the sense that they are obtained by the calculational approach based on Galois connections proposed by Cousot [28]. The non-existence of a known abstract domain for *pipeline analysis* constraints the calculational method of inducing a correct by construction abstract pipeline state-transformers. However, we present a semi-formal calculational approach to *pipeline analysis* in the sense that we provide a denotational semantics for pipeline behaviour, where state transitions are modelled using functional application [115].

Additionally, we compute a *program flow analysis* automatically at the machine-code level as the result of an instrumented *value analysis*, as stated in Contrib. (iv). According to Contrib. (iii), we also extend the generic data flow framework to support an interprocedural analysis based on the *functional approach* proposed in [128].

Finally, we make a case for Contrib. (x) to show that Haskell can be used where the mathematical complex notions underlying the theory of abstract interpretation can be easily, elegantly, and efficiently implemented and applied to the analysis of complex hardware architectures.

[1] Our concrete instruction semantics of ARM9 is based on the HARM virtual machine, an emulator for ARM programs that is written in Haskell: `http://hackage.haskell.org/package/HARM`

Our static analyzer is designed to be deployed in the supplier/consumer scenarios, typical in distributed embedded systems, where the theory of Abstract-Carrying Code (ACC) [9] can be used to implement a stand-alone and efficient safety-ensuring system. Along with the objective to provide a stand-alone verification mechanism of WCET estimates, special design requirements related to WCET analysis arise. For example, we cannot rely on manual annotations on the source code because the communication channel of the ACC framework only accepts the transmission of a *certificate* plus the corresponding *machine code* (as Fig. 1.3 illustrates). Consequently, the *program flow analysis* must be performed automatically at machine-code level and included in the certificate, so that program flow information can be verified at consumer sites.

More precisely, the program flow analysis computes the *capacity constraints* of the linear program, i.e. the upper bounds for the number of fixpoint iterations at every program point in the machine program. Using a proper semantic domain to specify program-flow information, bounds for loop iterations can be automatically computed using an instrumented version of the static analyzer described in Chapter 5. It follows that the labelled program states of Def. (5.5) must accommodate abstract invariants about both program-flow and machine-value information. However, this precludes the use of fixpoint convergence acceleration methods, in particular the widening/narrowing operators [31, 38], since complete loop unrolling is required by definition.

Nonetheless, the potential inefficiency of the static analyzer, in terms of analysis execution time directly associated to loop unrolling, is balanced by the gain in precision obtained when performing several static analyses simultaneously. For example, the `ait` WCET tool [49] computes the value, cache and pipeline analysis separately and then reuses the results of the previously computed analyses (in the specified order). Although each of the analysis is more efficiently computed, each static analysis must deal with the nondeterminism resulting from the approximative character of the abstract domains used by the other static analyses it depends on.

Alternatively, we compute the program flow, value, cache and pipeline analysis simultaneously, as illustrated by Fig. 6.1. The *value analysis* is performed during *pipeline analysis* using the "abstract instruction semantics" of the underlying hardware platform. According to the classifications *global low-level* analysis and *local low-level* analysis given in Chapter 4, our WCET static analyzer is designed so that it is able to compute increasing Kleene chains at the local low-level, while integrating results from the *value analysis* and the *global low-level analysis*. Every time that an instruction is fetched from the instruction memory, the resulting "execution facts" produced by the global low-level analysis, i.e. the *cache analysis*, are integrated into the local low-level analysis, i.e. the *pipeline analysis*. Additionally, the *pipeline analysis* also depends on the timing model of the underlying hardware platform.

This integrated solution is able to produce precise results by the fact that during fixpoint

iterations over instructions that are inside alternative paths or inside loops, the analyzer is able to detect precise information about the contents of the instruction cache and is able to detect if the instruction belongs to infeasible paths. Hence, the analysis of an instruction is a deterministic process as a consequence of the full loop unrolling. In fact, timing anomalies need to be considered only at those program labels where alternative paths join and they do not follow from the nondeterminism introduced by a previous separate cache analysis. In this work, we perform cache analysis in instruction caches only.

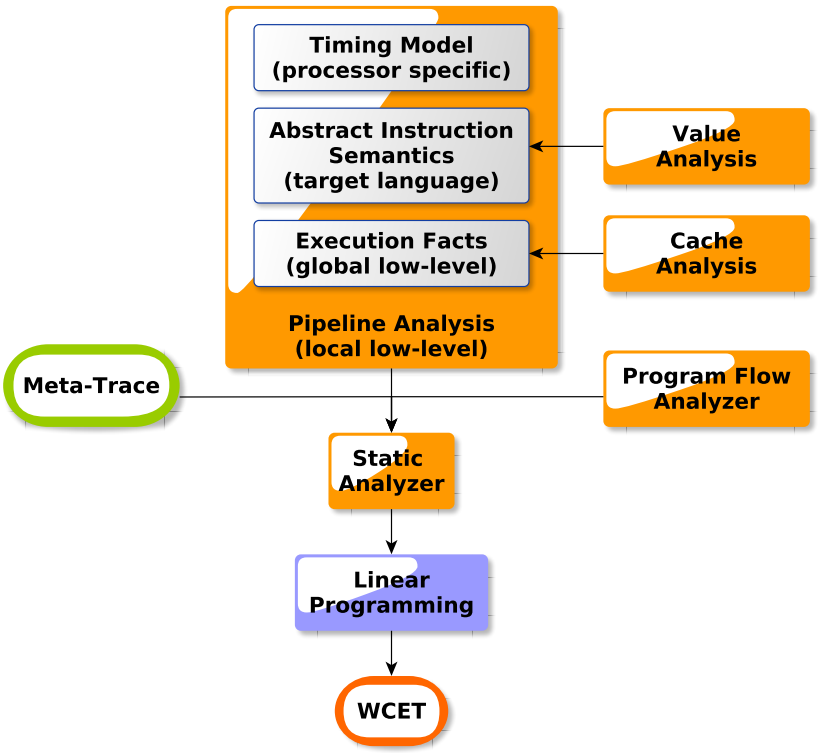


Figure 6.1: Overview of the Certifying Platform

6.1 Target Platform

Although the static analyzer presented in Chapter 5 is independent from the *defined language*, for the purpose of WCET analysis we need to provide a definition for the state transformers at the lower-level of the two-level denotational meta-language. Therefore, the choice of a particular hardware platform is required. We choose the ARM9 [125] target platform because it includes all the hardware components used by state-of-the art WCET tools, such as instruction and data caches, pipeline execution models, and also well-defined instruction semantics.

ARM9 is a 32-bit *Advanced RISC Machine* (ARM) architecture. RISC stands for *Reduced*

Instruction Set Computing and is a *Central Processing Unit* CPU design strategy that is based on the insight that simplified instructions can provide higher performance if that simplicity enables a much faster execution of each instruction. Relatively to its predecessor ARM7, the ARM9 architecture moved from a von Neumann architecture (Princeton architecture) to a Harvard architecture with separate instruction and data buses (and caches), significantly increasing its potential throughput. Over the years, successive generations of ARM processor cores are being massively used in mobile phones, handheld organizers (PDAs) and other portable consumer devices.

More technically, ARM9 is a Load/Store architecture, i.e. an architecture that only allows the memory to be accessed by *load* and *store* operations and requires that all values used by an *arithmetic logic unit* (ALU) operation to be present in registers. Space saving in instruction memories can be achieved by switching the "mode" of ARM to the Thumb 16-bit instruction set. Conditional execution is supported by allowing an instruction to be executed only when a specific condition has been satisfied. ARM9 has an orthogonal *Instruction Set Architecture* (ISA) in the sense that the instruction type and the addressing mode vary independently, i.e. an orthogonal instruction set does not impose a limitation that requires a certain instruction to use a specific register. The register file has the size 16 x 32 bits and instruction opcodes have a fixed width of 32 bits to ease decoding and pipelining.

6.2 Related Work

A reference work on the calculational design of semantics and static analyzers by abstract interpretation is the course given by Patrick Cousot at NATO International Summer School [28]. Considering a simple imperative language and its operational semantics, a pragmatic approach is taken to formally design and implement a generic abstract interpreter. The development is based on stepwise refinements and approximation of the fixpoint semantics and is decomposed in the four fundamental design options enumerated next (please note that second and fourth design options were already addressed in Chapter 5).

The first design option (1) is the approximation of the *reachability semantics* by the forward abstract invariant semantics. The second design option (2) is the definition of a system of fixpoint equations, by partitioning according to program points, through an isomorphic decomposition of the forward invariant semantics into local invariants [26]. The third design option (3) is the use of Galois connections to specify non-relational approximations of the local invariants by an abstract domain approximating sets of concrete values to get an attribute-independent abstract interpretation. Finally, the fourth design option (4) is the use of chaotic fixpoint strategies [33] for solving fixpoint equations in abstract domains that satisfy the ascending chain condition.

Particularly relevant for programming languages with conditionals is the fact that the weak-

est precondition and strongest postcondition *collection semantics* are no longer equivalent after approximation. Consequently, the generic abstract interpreter [28] is extended with *backward analyses* that can be combined iteratively with *forward analyses* [36].

Alternatively, Nielson shows in [94] how data flow analysis by abstract interpretation can be specified using a denotational approach, where a "store semantics" is systematically transformed into an "induced semantics" parametrized by a pair of adjoined functions or a Galois connection. One of the main conclusions is that the denotational approach is no less systematic than existing operational methods, although it is necessary to consider program transformations [97].

Recent work in the search for correct methods for automatically constructing a sensible abstract interpreter from a concrete semantics has been proposed in [89]. A two-step method is presented to convert a small-step concrete semantic semantics into a family of sound, computable abstract interpretations. The first step re-factors the concrete state-space by simultaneously eliminating recursive structure and applying a store-passing-style transformation of the concrete semantics. The second step uses inference rules to generate an abstract state-space and a Galois connection that allows the calculation of the "optimal" abstract interpretation.

6.3 Semantic Domains

The static analyzer combines the *value analysis* and the "micro-architectural" analysis. The latter is a term coined by the Compiler Design Group at the Saarbrücken University that assembles the *global low-level analysis* and the *local low-level analysis* in a single class of analyses. For the purposes of static analysis, the abstract domains and the corresponding abstract semantic transformers are determined according to the architecture configuration of the ARM9 microprocessor. The BNF specification of a subset of the ARM9 instruction set is given in Fig. 6.2.

$\langle RegisterName \rangle \rightarrow$ **R0** | **R1** | **R2** | ... | **R15** | **CPSR**
$\langle Op \rangle \rightarrow$ **Reg** $\langle RegisterName \rangle$ | **Con** *Word32* | **Rel** *Int* | **Bas** $\langle RegisterName \rangle$ *Word32*
$\langle Arith \rangle \rightarrow$ **Add** | **Sub** | **Mul**
$\langle Cmp \rangle \rightarrow$ **Cmp**
$\langle Br \rangle \rightarrow$ **B** | **Bne** | **Bgt** | **Blt** | **Beq**
$\langle Mem \rangle \rightarrow$ **Ldr** | **Str** | **Ldmfd** | **Stmfd**
$\langle Instr \rangle \rightarrow \langle Arith \rangle \langle Op \rangle \langle Op \rangle \langle Op \rangle \mid \langle Cmp \rangle \langle Op \rangle \langle Op \rangle \mid \langle Br \rangle \langle Op \rangle \mid \langle Mem \rangle \langle Op \rangle \langle Op \rangle$
$\langle Prog \rangle \rightarrow \langle Instr;^{+} \rangle$

Figure 6.2: BNF specification of a subset of the ARM9 machine language

A machine program $\langle Prog \rangle$ is a sequence of one or more instructions $\langle Instr \rangle$. Four types of instructions are considered. The first three instruction types are processed by the *arithmetic and logic unit* (ALU) funtional unit of the microprocessor and the fourth is the load/store

instruction type: (1) the instructions $\langle Arith \rangle$ read the values of the first two operands (constants (**Con** *Word32*) or registers (**Reg** $\langle RegisterName \rangle$)) and store the result on the third operand (**Reg** $\langle RegisterName \rangle$); (2) the comparison instruction $\langle Cmp \rangle$ reads and compares the values of two operands and updates the status flags in the register **CPSR**; (3) the branch instructions $\langle Br \rangle$ alter program execution by setting the "program counter", which is stored in the register **R15**, to a value based on a relative address given by the operand (**Rel** *Int*); and (4) the single data transfer instructions $\langle Mem \rangle$ exchanges data between memory and the register file and vice-versa.

The abstract CPU domain $\mathbb{C} \triangleq (R^\sharp \times D^\sharp \times M^\sharp \times C^\sharp \times P^\sharp)$ is a composite domain composed by an abstract register domain, $R^\sharp$, an abstract data memory domain $D^\sharp$, an abstract instruction memory domain, $M^\sharp$, an abstract instruction cache domain $C^\sharp$, and an abstract pipeline domain, $P^\sharp$. The abstract pipeline domain, $P^\sharp$, is defined as a collection of elements of an "hybrid" domain P, which is a composite domain defined as the cartesian product of the first four mentioned domains, $P \triangleq (R'^\sharp \times D'^\sharp \times M'^\sharp \times C'^\sharp)$. The definition of P in this way results from the fact that *value analysis* and "micro-architectural" analysis are performed simultaneously. This results in a single data-flow analysis defined on $P^\sharp$ and requires that the abstract states of all the allocated resources are available to perform the pipeline analysis of a single instruction. Consequently, the least upper bound between the elements of the top level domains $R^\sharp$, $D^\sharp$, $M^\sharp$ and $C^\sharp$ and the elements of the store buffers $R'^\sharp$, $D'^\sharp$, $M'^\sharp$ and $C'^\sharp$ is computed after the completion of the pipelining of every instruction.

An overview of the 5-stage Harvard pipeline architecture used by ARM9 is given in Fig. 6.3. The first stage (*Instruction Fetch*) depends exclusively on the instruction memory ("IM") because this stage is when the memory address given by the program counter is "fetched" from the instruction memory. In the second stage (*Instruction Decode*), the values of the operands of the fetched instruction are loaded from the register file ("Regfile"). At this point, store buffers must be created to carry out the processing of the "decoded" instruction. In the third stage (*Execution*), the "ALU" functional unit computes the hardware state after "executing" the instruction using the buffered operands. In the fourth stage (*Memory*), the data memory ("DM") is accessed, if that is required to execute the instruction. Finally, the final stage (*WriteBack*) is where the store buffers are loaded back into the globally shared register file ("Regfile"). The number of elapsed clock cycles in each stage are shown as "CC 1", "CC 2", etc., and indicate the clock cycles required to process a single instruction.

The Haskell definition of the abstract domain $\mathbb{C}$ is given by the datatype **CPU**. For the sake of readability, and since both the domains $M^\sharp$ and $C^\sharp$ are related to a single (cached) instruction memory, the datatype $\mathbf{I}^\sharp$ represents the entire structure of the instruction memory. We first give a brief introduction to the internal structure of each domain and then introduce the corresponding semantic transformers.

$$\textbf{data CPU} = \textbf{CPU}\ \{registers :: R^\sharp, dataMem :: D^\sharp, instrMem :: \mathbf{I}^\sharp, pipeline :: P^\sharp\}$$

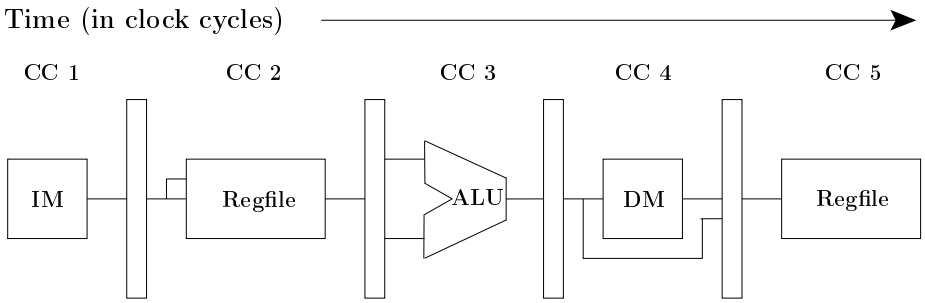


Figure 6.3: Graphical representation of an Harvard pipeline architecture

6.3.1 Register Abstract Domain

The register domain $R^\sharp$ denotes the ARM9 register file, also referred as *regfile*, and is denoted by the homonymous Haskell type $R^\sharp$. From the ARM9 BNF specification of Fig. 6.2, we can infer that the "abstract" environment of the register file maps 16 registers, syntactically phrased as (**Reg** ⟨*RegisterName*⟩), to their corresponding abstract values $\nu \in \mathbb{V}$. Accordingly, the Haskell type $R^\sharp$ is a fixed-size *Array* that maps an "index" to a abstract value. Each index is provided by the constructor **RegisterName** and the abstract type $\mathbb{V}$ is denoted by the datatype **RegVal**. Their corresponding definitions will be given later in Section 6.6.

type $R^\sharp$ = *Array* **RegisterName RegVal**

6.3.2 Data Memory Abstract Domain

The data memory domain $D^\sharp$ is a finite map from **Word32** memory addresses to abstract memory value $\nu \in \mathbb{V}$. Similarly to the register file $R^\sharp$, the homonymous type of $D^\sharp$ in Haskell is defined as:

type *Address* = **Word32**
type $D^\sharp$ = *Array Address* **RegVal**

6.3.3 Instruction Memory Abstract Domain

As already mentioned, the instruction memory domain consists of a main instruction memory domain $M^\sharp$ and an instruction cache domain $C^\sharp$. As opposed to the main data memory, the main instruction memory maps memory addresses to "opcodes". Additionally, the instruction cache maintains a set of cached values that consists in opcodes and their corresponding address in the main memory, associated with a tag within the scope of the cache. Both the abstract domains $M^\sharp$ and $C^\sharp$ are included in the instruction memory constructor $\mathbf{I}^\sharp$. Similarly to the data main memory, the main instruction memory domain is a fixed-size *Array* mapping *Address* indexes to **Opcode** values. The abstract instruction cache domain $C^\sharp$ is defined as a list (collection) of values of type *CacheLine*, as described in [48]. In its

turn, each cache line is modeled as a list of tagged **Opcode** values, where each Tag is an $Address$. All the memory addresses of the instruction main and cache memory are initialized with **Undefined**.

$$\textbf{type } M^\sharp = Array\ Address\ \textbf{Opcode}$$
$$\textbf{type } C^\sharp = [CacheLine]$$
$$\textbf{type } CacheLine = [MemoryBlock]$$
$$\textbf{type } MemoryBlock = (Tag, \textbf{Opcode})$$
$$\textbf{type } Tag = Address$$
$$\textbf{data Opcode} = \textbf{Opcode Word32} \mid \textbf{Undefined}$$
$$\textbf{data } \mathbf{I}^\sharp = \mathbf{I}^\sharp \{main :: M^\sharp, cache :: C^\sharp\}$$

6.3.4 Pipeline Abstract Domain

By design, the *value analysis* depends on the domains $R^\sharp$ and $D^\sharp$ and the *cache analysis* depends on the domain $C^\sharp$. Therefore, each fixpoint iteration of the *pipeline analysis* is defined in terms of a sequence of "hybrid" pipeline states P, each state consisting in the cartesian product of the domains $R'^\sharp$, $D'^\sharp$, $M'^\sharp$ and $C'^\sharp$. In [144] is stated that pipeline analysis by abstract interpretation is not a typical static program analysis because it employs both state traversal, as specified by a program-specific chaotic fixpoint strategy, and the least upper bound operations typically used in static analysis. In this sense, the pipeline analysis can be rather seen as an "history-sensitive" analysis, that requires the collection of all states encountered during fixpoint computation.

The reason for designing the abstract pipeline domain as a set of "hybrid" pipeline states is because there is no abstraction known to the WCET community for concrete timing properties [122]. Therefore, and according to the timing model of the ARM9 processor pipeline of Fig. 6.3, the domain P will be extended to include concrete timing properties, measured in *cycles per instruction* (CPI), in the cartesian product of the actual definition of P. In this way, the fixpoint algorithm is able to compute, for each instruction, an invariant on the hardware states that can occur whenever execution reaches that instruction, including all possible execution times in terms of number of cycles. After fixpoint stabilization, these execution times are given as input the WCET calculation component as the *capacity constraints* of the linear program.

For now, we define the abstract pipeline domain in Haskell using the constructor $P^\sharp$ as a list of "hybrid" pipeline states **P**. The Haskell definition of hybrid states is isomorphic to the cartesian product of $R^\sharp$, $D^\sharp$ and $\mathbf{I}^\sharp$. However, for sake of simplicity, the concrete information about the timing properties, e.g. the values "CC 1", "CC 2", "CC 3", shown in Fig. 6.3, are not yet included in the definition of **P**. A detailed description about the use of timing properties during *pipeline analysis* will be given latter in Section 6.8. Informally, we put forward that a concrete timing information will be included in the product of abstract

resource states in the definition of P.

$$\textbf{data P} = \textbf{P}\ \{registers :: R^\sharp, dataMem :: D^\sharp, instrMem :: \textbf{I}^\sharp\}$$
$$\textbf{newtype}\ P^\sharp = P^\sharp\ [\textbf{P}]$$

6.3.5 Abstract Semantic Transformers

Next, we introduce the semantic transformers required for WCET analysis. Generically, fixpoint semantics is specified by a pair $\langle L, F\rangle$, where L is the composite semantic domain equipped with a partial order ($\sqsubseteq$), an infimum element ($\bot$) and a partially defined least upper bound ($\sqcup$), and F is some monotone semantic transformer. Note that, in this context, F stands for a data semantic transformer on abstract values defined by the lattice L, rather than standing for a space of state-transformer functions on labelled program states Σ, as previously referred in Chapter 5.1.

The induction of "correct by construction" abstract semantic transformers is based on the Galois connection framework [37, Example 4.6], which establishes the relation $\langle \wp(L), \alpha, \gamma, L^\sharp\rangle$, between abstract values in $L^\sharp$ and sets of concrete values in L, where α and γ are the abstraction and concretization functions, respectively. Given that the collection semantics transformer, $F^\natural : \wp(L) \to \wp(L)$, is a complete join morphism and that the pair of functions $\langle \alpha, \gamma\rangle$ is a Galois connection, a *correct approximation* $F^\sharp = \alpha \circ F^\natural \circ \gamma$ is obtained by calculus. In this way, the fixpoint of $F^\sharp : L \to L$ is a sound over-approximation of the fixpoint of $F^\natural$ [37, Example 6.11].

The fixpoint semantics is divided in three distinct components: the first component $\langle (R^\sharp \times D^\sharp), F_V^\sharp\rangle$ is used for *value analysis*; the second component $\langle (C^\sharp, F_C^\sharp\rangle$ is used for *cache analysis*; and the third component $\langle P^\sharp, F_P^\sharp\rangle$ is used for *pipeline analysis*. The results of *cache analysis* are used by the pipeline analyzer that decides whenever a *cache miss penalty* must be added to the cycle-level semantics. In this way, as Fig. 6.1 illustrates, the transformer $F_P^\sharp$ invokes $F_E^\sharp$ and $F_C^\sharp$ in order to update its local copies of the resources $R'^\sharp$, $D'^\sharp$ and $C'^\sharp$. However, the life cycle of instruction inside the pipeline can also by affected by *pipeline hazards* [122]. This information is captured by the ARM9 pipeline timing model. In this thesis, we use a very simple and handwritten hardware timing model. Existing approaches are able to automatically generate this kind of models from the ARM9 VHDL specification [118].

The rest of this chapter is organized as follows. Using Fig. 6.1 as reference, we introduce the *program flow analysis* component in Section 6.4 and describe how it is closely related to the *value analysis*. Next, we describe the extensions necessary to the general data-flow framework presented in Chapter 5 to support *interprocedural analysis* in Section 6.5. Then, we describe in detail the calculational approach to the value and cache analysis, by giving correct by construction definitions of $F_V^\sharp$ and $F_C^\sharp$, in Sections 6.6 and 6.7 respectively, followed by the formalization of the abstract pipeline semantics $F_P^\sharp$ in Section 6.8.

6.4 Program Flow Analysis

This section describes our approach to automatic extraction of program flow information from source programs at machine-code level using abstract interpretation. In essence, *program flow analysis* is defined as a side effect of the *value analysis*, as enunciated in Contrib. (iv). The captured information is the number of fixpoint iterations performed in every edge of the dependency graph of the machine program. Intiutively, this mechanism is able to automatically extract loop bounds until the fixpoint algorithm stabilizes in the abstract value domain.

Tight WCET estimations require precise knowledge about program flow, namely the identification of *infeasible paths*, i.e. paths that can never be taken, and the minimal/maximal number of *iterations* in loops. Methods for program flow analysis include manual path annotations [104], loop unrolling [63], abstract interpretation [44, 57], and structural analysis of loops [61, 62]. Additionally, some application of program analysis provide a compact and efficient method for representing program flow information [43].

The *manual annotation* approach requires a considerable programming effort that is error-prone. Nonetheless, state-of-the-art tools, such as `aiT`, require user annotations to specify the targets of computed calls and branches and the maximum iteration counts of loops [2]. On the other hand, completely automatic approaches to program flow analysis can be found in [61, 44, 57, 62]. In general, these approaches depend on the conversion of the program flow representation into some form suitable for the WCET calculation, e.g. [80, 136].

An integrated approach is proposed in [84] to simultaneously perform program flow, cache, and pipeline analysis and calculation. Since this approach relies on architectural simulation techniques to simulate each program path on cycle-level timing models of the hardware platform, the complexity of the simulation process easily becomes prohibitive, especially in loop constructs. This problem is resolved using a *path-merging* technique aiming at the reduction of simulated program paths.

Although the integrated approach of [84] can be precise enough in the detection of "actual" feasible paths, which might be smaller than the statically allowed paths considered by abstract interpretation approaches [44, 57], it can be very inefficient for programs with a high number of loop iterations. Alternatively, the VIVU (Virtual Inlining of non-recursive functions and Virtual Unrolling of loops and recursive functions) approach presented in [86] is able to represent the program flow of loops as recursive functions. Instead of distinguishing all program paths, *path classes* are considered for which similar (timing) behavior is expected. This significantly reduces the number of program paths where to perform cache and pipeline analysis. The inconvenient of this approach is the possible loss in precision due to the analysis of unfeasible paths.

6.4.1 Declarative Approach

Since the calculation of the number of iterations in loops is in general equivalent to the halting problem, we must consider *all* possible executions by static analysis and reduce the computational cost by using abstractions, in order to compute the bounds of loop iterations. Our approach to program flow analysis is based on abstract interpretation and computes sound loop bounds and identifies infeasible paths as an *instrumented* value analysis, i.e. as a side effect of value abstract invariants computed at different program labels. Comparatively with [83] program flow analysis is, in fact, integrated with cache and pipeline analysis, but not with WCET calculation.

As illustrated by Fig. 6.1, the static analyzer uses the framework of abstract interpretation for the specification of the value, cache and pipeline data-flow analysis and proving them correct. The input of the static analyzer is a program's meta-trace, or a meta-program. This representation is a pre-compiled fixpoint algorithm, employing a program-specific chaotic fixpoint strategy. After the computation of the MFP fixpoint solution, the techniques of linear optimization are used to calculate the WCET. In the present Section, we define the *program flow analysis*, in particular how the generic data-flow analysis presented in Section 5.2 can be extended to integrate the program flow analysis as an instrumented abstract interpretation.

Instrumented abstract interpretation requires the definition of a cartesian product between the abstract invariants used for *value analysis*, which are denoted by Invs in Def. (5.4), and the invariants used for *program flow analysis*, here denoted by Flow in Def. (6.1). For automatic extraction of loop bounds the $\mathbb{L}$ domain is defined and equipped with an total order on positive integers. The element $\bot_{\mathbb{L}}$ denotes the number 0.

$$\mathbb{L} = \{\bot_{\mathbb{L}}\} \cup \{c \mid c \in \mathbb{N}^{+}\}$$
$$\sqsubseteq_{\mathbb{L}} = c_1 \sqsubseteq_{\mathbb{L}} c_2 \text{ iff } c_1 \leqslant c_2$$

The domain $\mathbb{L}$ is used in the definition of Flow as a map from an edge of the dependency graph into an element of $\mathbb{L}$. The program states Σ are then re-defined to include the Cartesian product (Invs $\times$ Flow) in the codomain. According to Def. (5.6), the domain of Flow, $\text{in}_P[\![P]\!]$ is a binary relation between the program label identifiers that can be found "at" the beginning of any transition in program P and the program label identifiers that can be found "after" the beginning of any transition in program P.

$$\text{Flow} \in \text{Prog} \mapsto \wp((\text{Lab} \times \text{Lab}) \hookrightarrow \mathbb{L}) \tag{6.1}$$
$$\text{Flow}[\![P]\!] \triangleq \text{in}_P[\![P]\!] \mapsto \mathbb{L}$$
$$\Sigma \in \text{Prog} \mapsto \wp(\text{Lab} \hookrightarrow (\text{Invs} \times \text{Flow})) \tag{6.2}$$
$$\Sigma[\![P]\!] \triangleq \text{at}_P[\![P]\!] \mapsto (\text{Invs}[\![P]\!] \times \text{Flow}[\![P]\!])$$

The Haskell definition corresponding to $\mathbb{L}$ is given by the constructor **Loop**. The partial order $\sqsubseteq_{\mathbb{L}}$ is defined using an instance of the type class *Ord*. Whence, the Haskell definition of *Flow* is straightforward. Finally, the program states defined in Haskell by means of the constructor (**St** a) are re-defined to include a record function *flow*, used to return the invariants map of type *Flow* contained in a program state.

```
data Loop = CountL Int | BottomL

instance Ord Loop where
  max (CountL a) (CountL b) = CountL (max a b)
  max (BottomL) (CountL b)  = CountL b
  max (CountL a) (BottomL)  = CountL a
  max (BottomL) (BottomL)   = BottomL

type Edges = (Lab, Lab)
type Flow = Map Edges Loop

data St a = St {label :: Lab, invs :: Invs a, flow :: Flow}
```

As specified by Def. (5.23), fixpoint iterations start with the $\Sigma_{\perp}$ undefined state. Every time the recursive operator $(\cdot \oplus \cdot)$ begins a new iteration using some data-propagation function, $f_{(k,l)}$, where k and l program label identifiers, which produces an output different from $\perp_{\Sigma}$, the actual loop count $c \in \mathbb{L}$ is incremented by 1 at the edge $k \rightarrow l \in E$. The loop iterations are incremented until the *value analysis* stabilizes. Since the static analyzer combines the results from value, cache and pipeline analysis in a single data-flow analysis, it is necessary to distinguish the fixpoint stabilization in each of the corresponding domains.

As already mentioned, the main advantage of the automatic extraction of program flow at machine level, when compared to manual annotations of program flow at source code level, is the simplicity of the processes. Nonetheless, this simplicity arises from our design option to use chaotic iteration strategies to mimic actual program executions. Therefore, our approach is a combination of loop unrolling with simulation in the abstract domain. Consider as an example the iteration strategy (5.13) given in Example 3. Intuitively, the loop bounds for nodes with depth 0 is always 1. For the nodes inside the recursive operator $[]^*$, the loop bounds correspond to the number of tail recursive calls inside the recursive operator $(\cdot \oplus \cdot)$ until fixpoint stabilization is achieved in the value abstract domain.

The next step is to define a semantic transformer over elements of $\mathbb{L}$ that can be lifted to the domain of program states and used in the refunctionalization of a dependency graph into an higher-order meta-program. Under the denotational assumption, the new state-transformer should be composed with the state-transformer previously defined in Section 5.2. In the same way, semantic state-transformers at denotational level are obtained by an abstraction of the relational semantics, using the right-image isomorphism based on Galois connections presented in [27].

Since there is a finite number of program labels for a given program P, we assume that the number of input-output relations $\tau \subseteq (\Sigma[\![P]\!] \times \text{Instrs}[\![P]\!] \times \Sigma[\![P]\!])$ is finite and equal to the

number of program instructions $\text{Instrs}[\![P]\!]$. This allows us to define the state vector $\Sigma[\![P]\!]$ as an ordered map from label identifiers to program invariants $\text{Invs}[\![P]\!]$, such that $\Sigma[\![P]\!] = \{\sigma_1, \ldots, \sigma_i, \sigma_j, \ldots, \sigma_n\}$, where n is the program label identifier of the last instruction of the program and σ_n is the abstract invariants map found in $\Sigma[\![P]\!]$ at the identifier n, which we abbreviate to Σ^n_P. As already mentioned in Section 5.1, when computing abstract fixpoints solutions using chaotic iteration strategies, the objective is the definition of a semantic state-transformer in the form of Def. (5.15), for which only one element of $\Sigma[\![P]\!]$ is updated in every fixpoint iteration.

The target denotational state-transformer $f_{\mathbb{C}\times\mathbb{L}}$, with the type $(\Sigma \mapsto \text{Instrs} \mapsto \wp(\Sigma))$, is obtained as an abstraction of the relational semantics, $\tau \subseteq (\Sigma[\![P]\!] \times \text{Instrs}[\![P]\!] \times \Sigma[\![P]\!])$, of program states, $\Sigma[\![P]\!]$, by the Galois connection $\langle \alpha_f, \gamma_f \rangle$.

$$\langle \wp(\Sigma \times \text{Instr} \times \Sigma), \subseteq \rangle \underset{\alpha_f}{\overset{\gamma_f}{\twoheadleftarrow\!\!\twoheadrightarrow}} \langle \Sigma \mapsto \text{Instr} \mapsto \wp(\Sigma), \dot{\leqslant} \rangle \tag{6.3}$$

Let σ_i and σ_j denote the abstract CPU states, of type $\mathbb{C}$, found in the $\text{Invs}[\![P]\!]$ invariants maps accessible from the components i and j of the state vector $\Sigma[\![P]\!]$, respectively. Also let $\delta_{i,j}$ denote the program flow information found in the $\text{Flow}[\![P]\!]$ invariants map at the edge $i \rightarrow j$. The abstraction function, α_f, and the concretization function, γ_f, are defined by:

$$\alpha_f(\tau) \triangleq \lambda\Sigma^i \cdot \{\Sigma^j \mid \forall \iota \in \text{Instrs}[\![P]\!] : \exists i, j \in \text{in}_P[\![\iota]\!] : \langle \Sigma^i, \iota, \Sigma^j \rangle \in \tau\} \tag{6.4}$$

$$\begin{aligned} \gamma_f(f_{\mathbb{C}\times\mathbb{L}}) \quad \triangleq \quad \{\langle (\Sigma^i, \iota, \Sigma^j) \rangle \quad \mid \quad & \forall \iota \in \text{Instrs}[\![P]\!] : \exists i, j \in \text{in}_P(\iota) : \\ & (\sigma_i, \delta_{i,j}) = \Sigma^i[i] \wedge (\sigma_j, \delta'_{i,j}) = \Sigma^j[j] \ \wedge \\ & (\sigma_j, \delta'_{i,j}) = f_{\mathbb{L}}(i,j) \circ f_{\mathbb{C}}(f_{(i,j)}\sigma_i)\} \end{aligned} \tag{6.5}$$

Therefore, and although the resulting abstract fixpoint semantics is an overapproximation of the optimal MOP fixpoint solution, it has the advantage that the fixpoint algorithm is specified at compile time by using both a *weak topological order* and a *chaotic iteration strategy* [20], and automatically compiled into expressions of our meta-semantic formalism. In this way, we compute sound fixpoint approximations at denotational level, using the pure declarative programming language Haskell.

The integration of the composition of the semantic state-transformers $f_{\mathbb{L}} \circ f_{\mathbb{C}}$ is performed inside the function *apply*, which is defined in the type class (*Abstractable a*) for refunctionalization. By the fact that program flow information requires the knowledge of the two program labels defined "at" and "after" a program instruction (5.6), we have to redefine the type signature of the function *apply* to include the "at" and "after" label identifiers, both of type *Lab*.

```
class Abstractable a where
  apply :: (Lab, Lab) → RelAbs (Invs a) → RelAbs (St a)
  lift :: Rel (St a) → RelAbs a → RelAbs (Invs a)
```

In the same way as in Section 5.2, the *refunctionalization* provided by the right-image isomorphism (5.21) is defined in Haskell by the function *refunct*. However, and although

the type of *refunct* is still **Rel** (**St** a)→*RelAbs* (**St** a), the function *apply* now requires the interpretation of the input label identifiers, because they contain the intensional information in Def. (6.1).

```
refunct :: (Abstractable a, Transition (Rel (St a))) ⇒ Rel (St a) → RelAbs (St a)
refunct r = let step = (dataFlow ∘ expr) r
                labels = ((point ∘ source) r, (point ∘ sink) r)
            in apply labels $ lift r step
```

Similarly to the previous definition, the new semantic state transformer, of type *RelAbs* (**St** a), is used to parametrize the static analyzer as a run-time entity in the context of the two-level denotational meta-language. However, besides updating the program invariants map (*Invs* a), it is also necessary to increment the loop iterations of a particular transition in the program flow map *Flow*, using as key a pair of labels, both of type *Lab*. In this way, the static analyzer computes the loop bounds contained in the constructor **Loop**, as a side effect of the analysis performed on abstract domain (**Env** a). For every program label, the last loop iteration computed before the fixpoint stabilization of the *value analysis* is taken as the upper loop bound.

The instrumentation of the value analysis requires the definition of an extra record function, *stable*. Given an abstract value of type (**Env** a), the return of the function *stable* is a list, which can be either empty or containing a combination of the elements provided by the induction data type **Stable**. Since the *program flow analysis* is defined as a side effect of the *value analysis*, the loop bounds invariants map *Flow* must be updated only during the fixpoint iterations for which the constructor **ValueStable** is not yet included in the output of *stable*.

```
data Env a = Env {value :: a, stable :: [Stable]}
data Stable = ValueStable | CacheStable | PipelineStable
```

The semantic state-transformer $f_{\mathbb{C}}$ is defined by the homonymous Haskell function $f_{\mathbb{C}}$. Given the state-transformer instance f, the update of the program abstract invariants in the input state s, is trivially done by applying the function f the invariants i, stored in the input state.

```
f_C :: RelAbs (Invs a) → RelAbs (St a)
f_C f s@St {invs = i} = s {invs = f i}
```

The semantic state-transformer $f_{\mathbb{L}}$ is defined by the homonymous Haskell function $f_{\mathbb{L}}$. The abstract invariants map previously computed by $f_{\mathbb{C}}$ is required to inspect the state of the fixpoint computation at the program label identifier "after" the instruction contained in the input-output relation r. The variable *fixed* is used to detect if the least fixpoint solution was already found in the abstract domain used for *value analysis*, by checking if the constructor **ValueStable** was found in the record function *stable* of the constructor (**Env** a). If the least fixpoint solution has already been reached, then the contents of the invariants map *Flow* are updated at the label *after* by means of the function *succ*. To this end, an instance of the type class *Enum* is defined for **Loop**. Otherwise, the invariants map remains unchanged.

```
f_𝕃 :: Edges → RelAbs (St a)
f_𝕃 rel s@St {invs = i, flow = f}
  = let fixed = elem ValueStable $ stable (i ! after)
        update = if fixed then id else succ
    in s {flow = adjust update (at, after) f}

instance Enum Loop where
  succ (CountL x) = (CountL (x + 1))
  succ (BottomL) = (CountL 1)
```

Finally, the generic instance of the function *apply*, defined in type class (*Abstractable* a), is simply the functional composition of the semantic state-transformers $f_{\mathbb{C}}$ and $f_{\mathbb{L}}$, as given by concretization function of Def. (6.5).

This is the reason why the isomorphic Galois connection $\langle \alpha_f, \gamma_f \rangle$ is so important to correctly define the process of *defunctionalization*, by means of γ_f, of the propagation functions (5.11) used by the MOP fixpoint algorithm and the posterior *refunctionalization*, by means of α_f, of the dependency graphs into the higher-order expressions of the two-level denotational meta-language. The purpose of the intermediate graph language defined in Section 5.3 is precisely the representation of functional compositions of propagation functions as connected sequences of input-output relations of type **Rel** (**St** a), which are then interpreted, using the function *derive*, in order to automatically generate a compositional fixpoint algorithm that is again based on the functional composition of functions with the type *RelAbs* (**St** a).

```
instance (Lattice a, Transition (Rel (St a)), Eq a) ⇒ Abstractable a where
  apply labels f = (f_𝕃 labels) ∘ (f_ℂ f)
```

Example 6. Simple program with a loop

Consider the simple C program of Fig. 6.4(a), where it is defined a "**while**" statement dependent on the value of "x". The objective of the *program flow analysis* is to automatically extract from the results of the *value analysis* the upper bound for the number of iterations of the "**while**" statement. The *value analysis* is performed on the code generated for the target platform ARM9. Fig. 6.4(b) shows the labelled relational semantics of the subset of instructions used to implement the "**while**" statement.

```
int main(void) {
  int x = 3;
  while (x>0) {
    x--;
  }
  return x;
}
```

(a) Source program

```
n10: b    16              :n6
 n8: ldr  r3, [fp, #-16]  :n7  {(".L3","main")}
 n9: sub  r3, r3, #1      :n8
n10: str  r3, [fp, #-16]  :n9
n11: ldr  r3, [fp, #-16]  :n10 {(".L2","main")}
n12: cmp  r3, #0          :n11
 n7: bgt  -20             :head_12
```

(b) Labelled relational semantics

Figure 6.4: Source program and the corresponding labelled relational semantics

The weak topological order of the machine program is obtained as an interpretation of the relational semantics yielding the following recursive iteration strategy:

$$\ldots\ 6\ 10\ 11\ [\underline{12}\ 7\ 8\ 9\ 10\ 11]^{*}\ \ldots \tag{6.6}$$

For the purpose of *program flow analysis*, we are interested in the fixpoint stabilization of the component inside the recursive operator $[\,]^{*}$. After inducing the weak topological, the next step is the automatic compilation of a meta-program that implements the previous chaotic iteration strategy using a composition of denotational state-transformers. A meta-program is an typed expression according to the combinators defined at the upper level of the two-level denotational meta-language defined in Section 5.2. Fig. 6.5 describes the meta-program that corresponds to the chaotic iteration strategy previously defined.

Each state-transformer is represented by its syntactic element, i.e. the instruction belonging to the corresponding input-output relation. Whence, the set of instructions that will be interpreted in the program states domain are the ones labelled from 7 to 12. However, the loop defined by the recursive operator $[\,]^{*}$ consists of a "head" label $\underline{12}$ and a sequence of labels that represent the body of the loop. In this way, the combinator $(\cdot \oplus \cdot)$ takes as the first argument the branch instruction '`bgt − 20`' and takes as the second argument the sequential composition $(\cdot * \cdot)$ of the instructions inside the loop body.

```
··· * (b 16) * (ldr r3, [fp, #-16]) * (cmp r3, #0) * ((bgt -20) ⊕ (ldr r3, [fp, #-16]) *
(sub r3, r3, #1) * (str r3, [fp, #-16]) * (ldr r3, [fp, #-16]) * (cmp r3, #0)) * ···
```

Figure 6.5: Meta-program derived from the iteration strategy (6.6)

Next, as an example, we present the details of the several fixpoint iterations performed until stabilization is reached for the label identifiers 10, 12 and 7. For label identifier 7, the computation of the abstract invariants for register '`R3`' is trivial in the sense that the first fixpoint iteration inside the loop starts with the invariant $[3, 3]$ and the lower bound of this invariant is reduced to $[0, 3]$ when the fixpoint condition is finished. The edges where the label identifier is used are $12 \rightarrow 7$ (feedback edge) and $7 \rightarrow 8$. Hence, the program flow information map is updated at (12,7) and (7,8) according to the semantics of $f_{\mathbb{C}}$.

Table 6.1: Program flow analysis for label 7 (4 fixpoint iterations)

Label	Loop Iteration 1		Loop Iteration 2		Loop Iteration 3	
	Value Analysis	Flow Analysis	Value Analysis	Flow Analysis	Value Analysis	Flow Analysis
7	R3 = [3,3]	(12,7) = 1 (7,8) = 1	R3 = [2,3]	(12,7) = 2 (7,8) = 2	R3 = [1,3]	(12,7) = 3 (7, 8) = 3

The state transformer $f_{\mathbb{C}}$ specifies that after each fixpoint iteration, the loop iteration count is *adjusted* with the successor of the previous count while the least fixpoint is not yet achieve for the value domain. Since $[0, 3]$ is the least fixpoint solution for '`R3`' at label identifier 7, the upper bound of loop iteration count is 4 at both edges (12,7) and (7,8). For program label identifier 10, we need to consider three edges: $6 \rightarrow 10$, $9 \rightarrow 10$ and $10 \rightarrow 11$. As can be inferred from Fig. 6.5, the instruction '`ldr r3, [fp, #-16]`' between the pair of label

identifiers (10,11) is also present outside the scope of the recursive combinator $(\cdot \oplus \cdot)$. Hence, the program flow information at (9,10) and (10,11) differs by 1 because the first edge to arrive at the identifier 10 was $6 \rightarrow 10$.

Table 6.2: Program flow analysis for label 10 (4 fixpoint iterations)

Label	Previous Iteration		Loop Iteration 1		Loop Iteration 2		Loop Iteration 3	
	Value Analysis	Flow Analysis	Value Analysis	Flow Analysis	Value Analysis	Flow Analysis	Value Analysis	Flow Analysis
10	R3 = [3,3]	(6,10) = 1 (10,11) = 1	R3 = [2,2]	(9,10) = 1 (10,11) = 2	R3 = [1,2]	(9,10) = 2 (10,11) = 3	R3 = [0,2]	(9,10) = 3 (10,11) = 4

Finally, for program label identifier 12, and although the instruction '`b 16`' between the pair of label identifiers (12,7) belongs to the scope of the recursive combinator $(\cdot \oplus \cdot)$, the upper bound for loop iterations is also 4 because the fixpoint stabilization condition is evaluated during the abstract interpretation of this instruction during the (extra) 4th fixpoint iteration. However, the fact that the upper bound of loop iterations at (7,8) is still 3, proves that the fixpoint stabilization condition was reached and that the Haskell fixpoint combinator *fix*, used in the definition of the combinator $(\cdot \oplus \cdot)$, terminates when the least fixpoint is found in the value domain. This example demonstrates how the *program flow analysis* is an instrumentation of the *value analysis*.

Table 6.3: Program flow analysis for label 12 (4 fixpoint iterations)

Label	Prev. Fixpoint Iteration		Loop Iteration 1		Loop Iteration 2		Loop Iteration 3	
	Value Analysis	Flow Analysis	Value Analysis	Flow Analysis	Value Analysis	Flow Analysis	Value Analysis	Flow Analysis
12	R3 = [3,3]	(11,12) = 1 (12,7) = 1	R3 = [2,3]	(11,12) = 2 (12,7) = 2	R3 = [1,3]	(11,12) = 3 (12, 7) = 3	R3 = [0,3]	(11,12) = 4 (12, 7) = 4

▲

6.5 Interprocedural Analysis

This section describes the application of the *function approach* to interprocedural analysis described in [128] to the WCET analysis of ARM9 programs as part of the Contrib. (iii). In general, a machine program includes one *branch-and-link* ('`bl`') instruction for each procedure call in the source code and one *load-registers-and-return* ('`ldmfd`','`ldmfa`', etc.) instruction for each procedure return. Since the notion of *weak topological order* can be extended to include procedures as new "components" contained in the hierarchical order [20], we simply define a proper chaotic iterations strategy in order to instantiate the required abstract state transformers. Afterwards, we use the higher-order combinator $(\cdot \oslash \cdot)$ to analyze interprocedural recursive compositions of abstract state transformers. In this way, the analysis of a procedure is seen as a "super-operation", defined as a composition of state transformers, which unified type is $\Sigma \rightarrow \Sigma$.

The main challenge in performing interprocedural analysis is the fact that procedure calls can be made from different call sites. Nonetheless, as stated in [128]: "it is always possible to transform a program with procedures into a procedureless program, by converting procedure calls and returns into ordinary branch instructions, monitored by an explicit stack." However, this inevitably overapproximates program flow because analysis information has to be propagated back to all possible call sites. Hence, this approach eventually leads to precision loss. A solution to this problem is to use *context information* in order to perform a context-sensitive interprocedural analysis.

A context-sensitive analysis considers the *calling context* when analyzing the target of a function call and also the *return context* once the procedure jumps back to the original call site. In [128] are introduced two techniques for performing interprocedural analysis. The main difference between them is the use of different graph models for the program being analyzed. The first approach is designated by *functional approach* and views procedures as collections of structured program blocks, such that each block is specified by an input-output relation. In the denotational sense, procedure calls are interpreted as "super operations", whose effect on context information can be computed using the constituent relations, and is valid even in the presence of recursion. A closely related approach is the in-line expansion of procedures [12].

The second approach is designated by *call-strings approach* and combines interprocedural flow analysis with the analysis of intraprocedural flow by turning a whole program into a single flow graph. However, context information is propagated along this path using "tags" that encode the history of procedure calls encountered during propagation. This makes interprocedural flow explicit and enables the determination of which part of the context information can be validly propagated through a procedure return, and which part has a conflicting call history that disallow such propagation.

An application of interprocedural analysis theory, in particular combined with the theory of abstract interpretation, is proposed in [86] for the data-flow analysis of loops. The essential idea is to treat loops as procedures by means of a transformation on the control flow graph such that program semantics are preserved. For the purpose of WCET estimation, the advantage of this approach follows from the fact that loops often iterates more than once and it is useful to distinguish the first iteration of a loop from the other ones. For some cache replacement policies, such as *first-in first-out* (FIFO), this distinction ascribes more precision to the results of cache behavior prediction and, consequently, leads to better pipeline analysis results.

More precisely, the approach proposed in [48, 86] is called VIVU and stands for a combination of *virtual inlining* of all non-recursive procedures and *virtual unrolling* of the first iterations of all recursive procedures, including loops. To this end, one execution context is instantiated to correspond to a path in the call graph of a program. The main characteristic of VIVO

is the distinction of *path classes*, in the sense that "paths through the call graph that only differ in the number of repeated passes through a cycle are not distinguished." However, since the number of loop iterations are not automaticallty determined, the results of analysis of loops with VIVU (e.g for the prediction of cache behavior) must be combined with the program flow information manually fed into the WCET framework in order to calculate a WCET estimate.

6.5.1 Declarative Approach

The integration of an interprocedural analysis into the meta-semantic formalism was already presented in Section 5.2 upon the definition of the interprocedural Haskell combinator (%). Our assumption is that any procedure might be recursive. Therefore, the analysis of a procedure is computed as the fixpoint of a functional that takes as argument an anonymous function that corresponds exactly to the expression $(f \,\%\, t)$, where f is the "body" of the procedure and t is the "return" relation of that procedure. In fact, this corresponds to a "super operation" that is composable, in the denotational sense, with the relation that previously made the procedure call and, posteriorly, with the relation immediately after the call site. For these reasons, our approach to interprocedural analysis is an application of the *functional approach* proposed in [128].

In order to analyze procedures as increasing Kleene chains [74], it is necessary to provide a fixpoint condition to stop the recursive application of the functional $(f \,\%\, t)$. For this purpose, we use the function *emptyStack* defined in the type class (*Iterable a*) introduced in Section 5.2. This effect of this function depends on context information, in particular, on the program label that originates the procedural call and on the program label from where the caller procedure continues execution. These two types of labels are distinguished in the weak topological order by the constructors **Call** and **Hook**, respectively. For example, in Fig. 6.6, the label 'call_11 {("foo","main")}' is instantiated using **Call**, and the label 'hook (6,"main")' is instantiated using **Hook**.

```
                        n1: mov   ip, sp              :root_0 {"main"};1
                            ...
call_11 {("foo","main")}: bl    24                    :n5
                        n7: mov   r3, r0              :hook (6,"main")
                            ...
              exit {"main"}: ldmfd sp, {r3,fp,sp,pc}  :n10
                       n12: mov   ip, sp              :root_11 {"foo"};2
                            ...
     exit {(".L4","foo")}: ldmfd sp, {r3,fp,sp,pc}  :n25
```

Figure 6.6: Labeled relational semantics of program 5.1 with procedure instructions

In a very summarized way, Fig. 6.6 shows the labelled relational semantics of a program with two procedures: "main" and "foo". Since the procedure "foo" can be invoked at different

call sites, the **Exit** label of the state "after" the return instruction '`ldmfd`' does not have an unique identifier that can be determined at compile time. Therefore, we introduce the notion of *execution context* in the definition of an abstract value to identify the procedure call sites and respective "hook" (jump back) labels. This is accomplished by adding two extra functions to the datatype (**Env** a), designated by *contexts* and *redirects*, respectively. The first function returns the history of procedure calls associated to a program state and the second function returns the corresponding stack of "hook" labels.

$$\textbf{data Env } a = \textbf{Env } \{value :: a, stable :: [\textbf{Stable}], contexts :: [\textbf{Label}], redirects :: [\textbf{Label}]\}$$

In those cases where context information is relevant for data-flow analysis, i.e. when the label of the sink state of a relation is an **Exit** label, it is necessary to use the context information stored in the program invariants map. For example, during the pipeline analysis using a particular iteration strategy, program labels are necessary to determine the current "program counter" of the machine program because they provide intensional information about the structure of the program. Therefore, once the "jump back" label is determined during pipeline analysis, it is necessary to store the updated context information in the program invariants map. This whole new process starts by re-defining the type class *Container*, which is used to read/write abstract values from/to the invariants map. At this phase, an instantiation of the type variable a is required.

Let the datatype **CPU** be an instance of the abstract value given by the type variable a denoting the hardware state of some microprocessor. Similarly to the datatype (**Env** a), the record function *calls* defined in **CPU** returns the list of procedure "call" labels. After instantiating the type variable a, it is necessary to define another instance of the type class *Container*, where the context information can be exchanged between the domain **CPU** and the domain of abstract values (**Env** a). The reason for this is that context information is required (and can be changed) during interpretations over **CPU** values, but needs also to be stored back into the program invariants map in every fixpoint iteration computing procedure calls and procedure returns.

$$\begin{aligned}\textbf{data CPU} = \textbf{CPU } \{&registers :: R^\sharp, dataMem :: D^\sharp, instrMem :: \mathbf{I}^\sharp, pipeline :: P^\sharp,\\ &calls :: [\textbf{Label}]\}\end{aligned}$$

The definition of the function *read* in the new instance of *Container* simply copies the context information stored inside an abstract *value* of type **Env CPU**, which is fetched from the program *invariants*, of type (*Invs* **CPU**), at the program *point* of the input *label*, into the list of procedure calls of the return hardware state, *cpu*.

Conversely, the definition of the function *store* has to specify the update of the *contexts* and *redirects* stored inside the nodes of a program *invariants* map. This update depends both on the *calls* of the input *cpu* hardware state and on type of labels that delimit the input relation *rel*, and is performed by the functions *newContext* and *deleteContext*. Afterwards, the *value* of a invariant's node is updated with the input *cpu* hardware state and the new context

information using the function *adjust*. In the same way, the list of *redirects* is updated upon a procedure return, i.e. when the *sink* label matches the constructor **Exit** when using the function *exit*.

```
instance (Lattice CPU) ⇒ Container (Invs CPU) CPU where
  read invariants label
        = let node = invariants ! (point label)
              cpu = value node
          in cpu {calls = contexts node}
  store rel cpu@CPU {calls = c} invariants
        = let (at, after) = (source rel, sink rel)
              context′ = deleteContext after $ newContext (at, after) c
              invs′ = adjust (updateValue cpu context′) (point after) invariants
          in if (exit ∘ sink) rel
               then adjust (updateRedirects calls) (point at) invs′
               else invs′
```

The function *newContext* updates context information upon a new procedure call. By definition, context information changes when the sink label of an input-output relation, holding as syntactical object a *branch-and-link* instruction (‘`bl`’), matches the constructor **Call**. This is detected by means of the function *call*, which is added to the definition of the type class (*Labeled a*). More precisely, the new context information corresponds to the “hook” label of the call site, and is easily determined by incrementing the *source* label of the relation holding the instruction ‘`bl`’.

```
newContext :: (Label, Label) → [Label] → [Label]
newContext (after, at) contexts
  = case call after of
      False → contexts
      True → (succ (at)) : contexts
```

Example 7. Simple program with a procedure call

For example, in the relational semantics of Fig. 6.6, the instruction ‘`bl` *24*’ is delimited by the label descriptions ‘n5’ and ‘call_11 (“foo”, “main”)’. Since the label *after* the instruction is a *call* label, the context information consists in the “hook” label described as ‘n6’, which is precisely the successor of ‘n5’, i.e. the label *at* (or before) the instruction. However, the label identifiers 5 and 6 belong to different components in the weak topological order and are not sequentially organized. In fact, the weak topological order of Def. (5.10) shows that the label point 11 starts the *inlining* of the procedure ”foo” and the label point 6 is the “hook” site corresponding to the particular ‘call_11’. ▲

On the other hand, the function *deleteContext* updates the context information when a procedure returns, i.e. when the sink label *after* is an *exit* label. In the affirmative case, the first element in the stack of procedure call labels is popped out using, for that purpose,

the function *tail*. Therefore, in analogy with a FIFO stack, procedure calls are modelled in such a way that the return of each procedure must be made some instructions after the call of that same procedure.

```
deleteContext :: Label → [Label] → [Label]
deleteContext after contexts
  = case exit after of
      False → contexts
      True → tail contexts
```

As already mentioned, the functions *call* and *exit*, required by the interprocedural analysis, are defined in the type class (*Labeled a*), which is instantiated by the datatype **Label**. The additional function *setId* is defined in order to update the *labelId* of a label identifier.

```
class Labeled a where
  point :: a → Lab
  head  :: a → Bool
  call  :: a → Bool
  hook  :: a → Bool
  exit  :: a → Bool
  setId :: a → Lab → a
```

In the same way, the type class (*Abstractable a*) requires changes when lifting the semantic transformer on program invariants to program states. The reason follows from the fact that **Exit** labels have unknown point identifiers, which have to updated using context information so that the "super operation" of the procedure can be functionally composed with the state-propagation functions of the "caller". Indeed, this process is compliant with the requirements of fixpoint computations using chaotic iteration strategies induced by the weak topological order of programs. The next task is to modify the "after" state of a relation so that it becomes a new state, with the same program invariants and program flow, but labelled with an **Hook** label, which is dynamically determined during interprocedural analysis using the function *returnContext*.

```
returnContext :: (Transition r) ⇒ r → CPU → r
returnContext rel cpu@CPU {calls = (c : cs)}
  = let after = sink rel
    in adapt rel $ case exit after of
                     False → after
                     True → setId after (point c)
```

In order the modify the *labelId* of a label identifier, i, with a *new* integer value, it is necessary to provide an instance of the type class (*Labeled a*). The function of main interest is *setId*, which simply updates the *labelId* of the identifier i of an **Exit** label. The reader is referred to the Haskell prototype for the definitions of the remaining functions defined in the type class *Labeled*.

```
instance Labeled Label where
  setId (Exit i) new = Label i {labelId = new}
```

After modifying the *labelId* of the *sink* label of the input relation using the function *setId*, it is necessary to *adapt* the "after" state of that relation. As usual, functions manipulating relations are defined in the type class (*Transition* a). The new function *adapt* simply updates the "after" state, b, with the provided *label*, l. The remaining definitions of the functions *sink*, *source* and *expr* are also given.

```
class Transition a where
  sink   :: a → Label
  source :: a → Label
  expr   :: a → Expr
  adapt  :: a → Label → a

instance Transition (Rel (St a)) where
  sink   (Rel (a, _, _))   = label a
  source (Rel (_, _, b))   = label b
  expr   (Rel (_, i, _))   = i
  adapt  (Rel (b, e, a)) l = Rel (b {label = l}, e, a)
```

Finally, we re-define the instance of the type class (*Abstractable* a). First, we re-define the function *lift* so that the relation passed as argument to the *chaotic* fixpoint function (see Section 5.2) has the proper surrounding labels when iterating over procedure return instruction. This implies the "inlined" relation to be included in the return type of the function *lift*. To this end, the new "extended" types (*ExtInvs* a) and (*ExtSt* a) are defined as pairs that include the "inlined" relation label identifiers, aside with the respective types (*Invs* a) and (**St** a). Additionally, the "extended" function types (*ExtInvsAbs* a) and (*ExtStInvs* a) are defined to substitute the type (*RelAbs* a).

```
type ExtInvs a = (Edges, Invs a)
type ExtSt a = (Edges, St a)
type ExtInvsAbs a = Invs a → ExtInvs a
type ExtStAbs a = St a → ExtSt a
```

Since the function *refunct*, which is the Haskell definition of the relational abstraction of input-output state relations into denotational state-transition functions, depends on both functions *lift* and *apply*, it is also required to change the type signature of these functions. Firstly, the return type of *lift* has the functional extended type (*ExtInvsAbs* a) because the "after" state of the inlined relation may have been updated will a new interprocedural context. Secondly, the input argument must be re-defined to receive information about the inlined relation.

```
class Abstractable a where
  apply :: ExtInvsAbs a → RelAbs (St a)
  lift :: Rel (St a) → RelAbs a → ExtInvsAbs a
```

According, it is required a new instance of type class (*Abstractable* a), where the type variable a is instantiated by the hardware constructor **CPU**, so that the definition of the function

refunct is preserved. The only difference in the re-definition of *lift* is the inclusion of the interprocedural label information produced by *returnContext* in the returned pair. On the other hand, the re-definition of *apply* consists on the fact the label identifiers required for *program flow* analysis is directly provided to the state transformer $f_{\mathbb{L}}$. by the state transformer $f_{\mathbb{C}}$.

```
instance (Lattice CPU, Transition (Rel (St CPU))) ⇒ Abstractable CPU where
  apply f = f_L ∘ (f_C f)
  lift r f invs = let s' = f $ read invs (source r)
                      r' = returnContext r s'
                      labels = ((point ∘ source) r', (point ∘ sink) r')
                  in (labels, chaotic r' invs s')
```

The re-definition of the state transformers are only the consequence of the use of new "extended" types. Whence, the definitions of $f_{\mathbb{C}}$ and $f_{\mathbb{L}}$ have a few minor changes:

```
f_C :: ExtInvsAbs a → ExtStAbs a
f_C f s@St {invs = i} = let ext = f i
                        in (fst ext, s {invs = snd ext})

f_L :: ExtSt a → St a
f_L (pair, s@St {invs = i, edges = e}) = let fixed = elem ValueStable $
                                                stable (i ! (snd pair))
                                             update = if fixed then id else succ
                                         in s {edges = adjust update pair e}
```

6.6 Value Analysis

The value analysis is based on the *interval abstraction* introduced by the Cousots in [31, 32]. The interval abstraction is applied to the analysis of the hardware components that store 32-bit values, more specifically, the concrete register file, R, and the concrete data memory, D. However, there are some exceptions in the register file: cases where a general-purpose register is designed to store some sort of "control" information. These registers are the "frame pointer" register **R11**, the "intra-procedural-call scratch" register **R12**, the "stack pointer" register **R13**, the "branch-and-link" register **R14**, the "program counter" register **R15**, and, finally, the "status" register **CPSR**.

Therefore, comparatively to the definition given in Section 6.3.1, we now design the abstract register file, $R^\sharp$, in such a way that the values of the general purpose registers **R0-R10** are abstracted into interval values $\nu \in \mathbb{V}$, and the rest of the register values are kept in the concrete domain of 32-bit values $\mathbb{W}$, is to improve the precision of the analysis. For example, in the case of the register **R15**, this is indeed necessary because instruction *cache analysis* is performed simultaneously with the *value analysis*. In practice, when the chaotic fixpoint algorithm is computing an iteration over of the effect of a particular instruction in

the machine code, it is first required to fetch this instruction from the instruction memory in order to classify the memory access as a "cache miss" or a "cache hit". Therefore, the memory address stored inside **R15** must be a concrete, 32-bit value, so that the fetching process can be deterministic.

As a second example, consider the "stack pointer" register **R13**. Compared to the register **R15**, the reason to keep its value in the concrete domain is different but also related to precision. As will be described in Section 7, it is possible to perform WCET verification at source-code level when compiler debug information [134] is able to establish a correspondence between the value of a source program variable to the memory address that stores that value at machine-code level. Therefore, the possibility to know exactly such memory address improves the precision of verification process.

We state that although we do not perform abstract interpretation for "control" registers, their values are updated accordingly to the chaotic fixpoint strategy induced for a particular program. Since the chaotic fixpoint algorithm is flow sensitive, the referred registers are always updated with values computed during the "last" fixpoint iteration. Consequently, the least upper bound operator defined for the abstract domain of register values, has to reflect which type of information is "lost" and how some of this information can still be approximated across fixpoint computations.

6.6.1 Related Work on Interval Abstraction

The interval abstraction introduced in [31, 32] are defined for integer values. New challenges arise when is necessary to perform interval arithmetic on other kind of domains, e.g. inside the domain of unsigned 32-bit values. The approach described in [65] is a known example of how optimal solutions can be found to deal with the difficulties found in performing interval arithmetic using real numbers. An example of the general applicability of interval analysis to program analysis is given in [51], where interval analysis is used to evaluate conditional expressions in the general polynomial form in order to guide the process of optimizing transformation of loops containing nested conditional blocks. Section 6.6.4.2 demonstrates the utility of the interval abstraction to perform backward abstract interpretation of loops in order to evaluate conditional instructions in the abstract domain.

Other applications of the interval abstraction in the context of abstract interpretation include the analysis of C-variables presented in [45] and the formalization of the abstraction using the Galois connection framework given [73]. Both approaches identify the fundamental drawback of applying abstract interpretation to interval analysis, which is the fact that the fixpoint algorithm is not guaranteed to terminate because there exist infinitely increasing chains of intervals. The original solution to this problem was introduced in [31] by means of *widening* and *narrowing*. As previously mentioned, our approach does not implement these fixpoint

acceleration operators because we assume that, for the purpose of WCET estimation, only programs that do terminate can be analyzed. Technically, this is a requirement because we perform non-virtual loop unrolling. Nonetheless, it should be pointed out that, by definition, the general data flow framework presented in Chapter 5 does not preclude the use of such operators.

6.6.2 Concrete Semantics

The domain of concrete register values is denoted by $R \triangleq \mathbb{N} \mapsto \mathbb{W}$, where $\mathbb{N} \triangleq \langle RegisterName \rangle$, is the set of registers names and $\mathbb{W}$ is the domain of 32-bit machine values. A register set $\rho \in R$ records the value $\rho(n)$ of the register names $n \in \mathbb{N}$. Similarly, the domain of data memory values $D \triangleq \mathbb{A} \mapsto \mathbb{W}$ is a map from 32-bit address values $\mathbb{A}$ to $\mathbb{W}$. Updates to the environment ρ are done in Haskell by means of the *setReg* and the contained values $\rho(n)$ accessible by means of the function *getReg*:

```
type R = Array RegisterName Word32
setReg :: R → [(RegisterName, Word32)] → R
setReg = ( // )
getReg :: R → RegisterName → Word32
getReg = ( ! )
```

In general, the interpretation of an instruction *Instr* takes as the environment the current state, consisting in the product $(R \times D)$ of the 32-bit valued maps of type R and D, and returns a new environment of the same type. In this sense, the instruction semantics is defined by the semantic transformer F_V:

$$F_V \in Instr \mapsto (R \times D) \mapsto (R \times D) \tag{6.7}$$

Consider as an example the instruction **Add** when the values of two register operands are added. The concrete functional behavior can be easily defined in Haskell by means of the function f_V (note that, is this case, the input data memory map d does not produce change):

```
fV :: Instr → (R, D) → (R, D)
fV (Add (Reg reg1) (Reg reg2) (Reg reg3)) (r, d)
  = let v2 = getReg r reg2
        v3 = getReg r reg3
    in (setReg reg1 (v2 + v3) r, d)
```

6.6.3 Abstract Domain

The abstract interpreter used for *value analysis* is obtained by induction as an abstraction of the instruction semantics using intervals of 32-bit values as the abstract domain. Intervals are abstract values $\nu \in \mathbb{V}(\mathbb{W})$, in such a way that $\mathbb{V}$ can be parametrized to contain the

32-bit values defined in the domain $\mathbb{W}$. Hence, the value analysis is defined in terms of the lattice $\mathbb{V}(\mathbb{W})$, with the least upper bound operator ($\sqcup^\sharp_\nu$) and the greatest lower bound operator ($\sqcap^\sharp_\nu$):

$$\mathbb{V}(\mathbb{W}) = \{\bot^\sharp\} \cup \{[l,u] \mid l \in \mathbb{W} \cup \{-\infty\} \wedge u \in \mathbb{W} \cup \{+\infty\} \wedge l \leqslant u\}$$

$$[l_1,u_1] \sqcup^\sharp_\nu [l_2,u_2] = [min(l_1,l_2), max(u_1,u_2)] \tag{6.8}$$

$$[l_1,u_1] \sqcap^\sharp_\nu [l_2,u_2] = [max(l_1,l_2), min(u_1,u_2)] \tag{6.9}$$

The Haskell definition of abstract domain $\mathbb{V}(\mathbb{W})$ is given by the constructor **AbstVal** using as argument an *Interval* stored inside an analysis general-purpose register (**R0-R10**). The limit values $-\infty$ and $+\infty$ correspond to the minimum and maximum values of a signed 32-bit word. The registers used by the static analyzer to store control information (**R11-R15**) use the constructor of concrete **Word32** values **ConcVal**. For the reasons previously mentioned, an explicit constructor of the abstract value of the register **CPSR** is not yet present in the definition of **RegVal**, but it will be included in the future re-definition of **RegVal**, given in Section 6.6.4.2, when the process of *backward abstract interpretation* of conditional instructions is described. For now, we assume that the **CPSR** register stores a concrete 32-bit value.

```
data RegVal = AbstVal Interval | ConcVal Word32 | Bottom
type Interval = (Word32, Word32)
```

Although the inductive Haskell type constructor **RegVal** is able to "algebraically" compose the constructors **AbstVal** and **ConcVal**, two elements of these two different types are not comparable by design. Therefore, the inductive constructor **RegVal** does not denote a lattice, in the sense that is not a complete partial order. However, using the *coalesced* domain definition given in [13], the same constructor can indeed denote the coalesced domain *Interval* + **Word32**, lifted with a common undefined element **Bottom**. Nonetheless, the instance of the type class (*Lattice a*) must be defined for the co-product datatype **RegVal**.

The next step is to define the lattice of interval values, $\nu \in \mathbb{V}(\mathbb{W})$. Despite the fact that the elements of $\mathbb{W}$ are unsigned 32-bit values, we are still interested in using the interval arithmetics for integer values [45, 73]. To this end, we have defined the function *toInt32* that converts a 32-bit unsigned word into an **Integer**. This conversion is straightforward: the maximum number of an 32-bit unsigned word is 4294967296; an unsigned word is converted into a negative or positive integer by dividing the maximum number by 2 and subtracting 1, which gives 2147483647, and taking the difference to the maximum number. The Haskell function *fromIntegral* is used to convert the type **Word32** into the type **Integer**.

```
toInt32 (w :: Word32) = let w' = fromIntegral w :: Integer
                        in if w' > 2147483647
                           then w' - 4294967296
                           else w'
```

The instantiation of the type class *Lattice* for the type *Interval* uses the function *toInt32* so that the join and meet operators defined in (6.8) and (6.9) can be directly implemented. For this purpose, the functions *meet* was added to the definition of (*Lattice* a). Afterwards, the results are converted back to the **Word32** type by means of the function *fromIntegral*. Note that the following instance of *Lattice* does not implement the function *bottom* because the undefined element is only defined for the coalesced domain **AbstVal** + **ConcVal**, and not for each one of the composed domains.

```
instance Lattice Interval where
  join (a, b) (c, d)
    = let (a', b') = (toInt32 a, toInt32 b)
          (c', d') = (toInt32 c, toInt32 d)
      in (fromIntegral (min a' c') :: Word32, fromIntegral (max b' d') :: Word32)
  meet (a, b) (c, d)
    = let (a', b') = (toInt32 a, toInt32 b)
          (c', d') = (toInt32 c, toInt32 d)
      in (fromIntegral (max a' c') :: Word32, fromIntegral (min b' d') :: Word32)
```

On the one hand, assuming that the chaotic fixpoint strategy allows the static analysis to mimic the program execution, the partial order $\sqsubseteq^{\delta}_{\mathbb{W}}$ on elements of the *concrete* domain $\mathbb{W}$ is induced by the coeficient δ used in definition (5.15), which indicates the number of fixpoint iterations already performed, so that when we write that $a \sqsubseteq^{\delta}_{\mathbb{W}} b$ implies that $a \sqcup^{\delta}_{\mathbb{W}} b = b$, it means that a is a value computed during iteration δ and b is a value computed during iteration $\delta + 1$. The same applies to the greatest lower bound operator $\sqcap^{\delta}_{\mathbb{W}}$. On the other hand, the *abstract* domain $\mathbb{V}(\mathbb{W})$ has the least upper bounds of Def. (6.8) and the greatest lower bounds of Def. (6.9). Finally, the instance of *Lattice* for the coalesced inductive constructor **RegVal** is (for sake of simplicity the definitions of *join* and *meet* involving the **Bottom** constructor are omitted here):

```
instance Lattice RegVal where
  bottom = Bottom
  join (ConcVal a) (ConcVal b) = ConcVal b
  join (AbstVal a) (AbstVal b) = AbstVal (join a b)
  meet (ConcVal a) (ConcVal b) = ConcVal b
  meet (AbstVal a) (AbstVal b) = if disjoint a b
                                 then Bottom
                                 else AbstVal (meet a b)
```

The definition of the function *meet* requires an auxiliary function designated by *disjoint* that returns **True** if two intervals given as inputs do not intersect. In such cases, the greatest lower bound on the two intervals is, by definition, **Bottom**.

```
disjoint :: Interval → Interval → Bool
disjoint (a, b) (c, d)
  = let (a', b') = (toInt32 a, toInt32 b)
        (c', d') = (toInt32 c, toInt32 d)
    in max a' c' > min b' d'
```

As expected, the Galois connection used for *value analysis* only considers the subset of registers that store abstract values, $\nu \in \mathbb{V}(\mathbb{W})$. Therefore, from the previously defined set of register names $\mathbb{N}$, we now define a subset $\mathbb{N}_\mathbb{V}$ of register names to each the interval abstraction applies. Next, we formulate the correctness of the interval abstraction as a Galois connection [73]. The approximation of sets of concrete values, defined as elements of the powerset lattice $\wp(\mathbb{W})$, into intervals inside $\mathbb{V}(\mathbb{W})$, is defined by the Galois connection $\langle \alpha, \gamma \rangle$:

$$\langle \wp(\mathbb{W}), \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathbb{V}(\mathbb{W}), \sqsubseteq^\sharp_\nu \rangle \tag{6.10}$$

The definitions of α and γ for the interval abstraction are:

$$\alpha(S) = \begin{cases} \bot^\sharp_\nu, & \text{if } S = \emptyset \\ [a, b], & \text{if } min(S) = a \text{ and } max(S) = b \end{cases} \tag{6.11}$$

$$\gamma(i) = \begin{cases} \emptyset, & \text{if } i = \bot^\sharp_\nu \\ \{w \in \mathbb{W} \mid a \leqslant w \leqslant b\}, & \text{if } i = [a, b] \end{cases} \tag{6.12}$$

The previous formal definitions of α and γ are in direct correspondence to their Haskell definitions, *abst* and *conc*, respectively.

```
abst :: [Word32] → RegVal
abst [] = Bottom
abst s  = AbstVal (minimum s, maximum s)

conc :: RegVal → [Word32]
conc Bottom = []
conc (AbstVal (l, u)) = [l .. u]
```

The abstract definition $R^\sharp$ of the concrete register set R is obtained by the composition of two abstractions: the first abstraction is called *non-relational* because all possible relationships between the register values are lost in the abstraction [30]; the second abstraction is called *codomain* abstraction as it based on the Galois connection (6.10), defined for content of each particular register.

The non-relational abstraction is defined by the Galois connection $\langle \alpha_r, \gamma_r \rangle$ and approximate properties of register sets by ignoring relationships between the possible values associated to register names:

$$\langle \wp(\mathbb{N}_\mathbb{V} \mapsto \mathbb{W}), \subseteq \rangle \underset{\alpha_r}{\overset{\gamma_r}{\leftrightarrows}} \langle \mathbb{N}_\mathbb{V} \mapsto \wp(\mathbb{W}), \dot{\subseteq} \rangle. \tag{6.13}$$

This abstraction approximates sets of concrete maps $R^\natural \triangleq \wp(\mathbb{N}_\mathbb{V} \mapsto \mathbb{W})$ to a non-relational collecting concrete semantics $R^\natural_r \triangleq \mathbb{N}_\mathbb{V} \mapsto \wp(\mathbb{W})$. Given the register set $\rho \in R$, the definitions of the Galois connection is the following:

$$\alpha_r(R^\natural) = \lambda n \in \mathbb{N}_\mathbb{V} \bullet \{\rho(n) \mid \ \rho \in R^\natural\}$$
$$\gamma_r(R^\natural_r) = \{\rho \mid \forall n \in \mathbb{N}_\mathbb{V} : \rho(n) \in R^\natural_r(n)\}$$

where the pointwise ordering $\dot{\subseteq}$ is defined by:

$$R_r^\natural \dot{\subseteq} R_r'^\natural \triangleq \forall n \in \mathbb{N} : R_r^\natural(n) \subseteq R_r'^\natural(n).$$

The codomain abstraction is defined by the Galois connection $\langle \alpha_c, \gamma_c \rangle$ and approximate the codomain of $R_r^\natural$, designated as $R_\nu^\sharp$, using the Galois connection $\langle \alpha, \gamma \rangle$ of Def. (6.10):

$$\langle \mathbb{N}_\mathbb{V} \mapsto \wp(\mathbb{W}), \dot{\subseteq} \rangle \underset{\alpha_c}{\overset{\gamma_c}{\leftrightarrows}} \langle \mathbb{N}_\mathbb{V} \mapsto \mathbb{V}(\mathbb{W}), \dot{\sqsubseteq}_\nu^\sharp \rangle \tag{6.14}$$

where

$$\begin{aligned} \alpha_c(R_r^\natural) &\triangleq \alpha \circ R_r^\natural, \\ \gamma_c(R_\nu^\sharp) &\triangleq \gamma \circ R_\nu^\sharp, \\ R_\nu^\sharp \dot{\sqsubseteq}_\nu^\sharp R_\nu'^\sharp &\triangleq \forall n \in \mathbb{N}_\mathbb{V} : R_\nu^\sharp(n) \sqsubseteq_\nu^\sharp R_\nu'^\sharp(n). \end{aligned}$$

Therefore, the abstract register set $R_\nu^\sharp \triangleq \mathbb{N}_\mathbb{V} \mapsto \mathbb{V}(\mathbb{W})$ is defined to be a complete lattice for the pointwise ordering $\dot{\sqsubseteq}_\nu^\sharp$. Finally, the composition of the non-relational and codomain abstractions is given by the Galois connection $\langle \dot{\alpha}, \dot{\gamma} \rangle$:

$$\langle \wp(\mathbb{N}_\mathbb{V} \mapsto \mathbb{W}), \subseteq \rangle \underset{\dot{\alpha}}{\overset{\dot{\gamma}}{\leftrightarrows}} \langle \mathbb{N}_\mathbb{V} \mapsto \mathbb{V}(\mathbb{W}), \dot{\sqsubseteq}_\nu^\sharp \rangle \tag{6.15}$$

where

$$\begin{aligned} \dot{\alpha}(R^\natural) &\triangleq \alpha_c \circ \alpha_r(R^\natural) \\ &= \lambda n \in \mathbb{N}_\mathbb{V} \bullet \alpha(\{\rho(n) \mid \rho \in R^\natural\}), \\ \dot{\gamma}(R_\nu^\sharp) &\triangleq \gamma_r \circ \gamma_c(R_\nu^\sharp) \\ &= \{\rho \mid \forall n \in \mathbb{N}_\mathbb{V} : \rho(n) \in \gamma(R_\nu^\sharp(n))\}. \end{aligned}$$

Finally, we have to define the abstract register domain for the entire set of register names $\mathbb{N}$. Let $\mathbb{N}_\mathbb{W} = \mathbb{N} \setminus \mathbb{N}_\mathbb{V}$ be the set of registers storing "concrete" control information, obtained as the set difference between $\mathbb{N}$ and $\mathbb{N}_\mathbb{V}$. Now let $R_w \triangleq \mathbb{N}_\mathbb{W} \mapsto \mathbb{W}$. The coalesced (coproduct) type constructor $\mathbf{AbstVal} + \mathbf{ConcVal}$ is then formally defined the disjoint union of the two previously defined maps, $R_\nu^\sharp$ and R_w: $R^\sharp \triangleq R_\nu^\sharp \biguplus R_w$. Hence, the Haskell definition of $R^\sharp$ given in the Section 6.3.1 must be re-defined to include in the domain all the possible sorts of abstract/concrete values:

type $R^\sharp$ = *Array* **RegisterName RegVal**

6.6.4 Calculational Design

6.6.4.1 Forward Abstract Interpretation of the Add instruction

The design of an abstract interpretation $F_V^\sharp$ is based on the approximation of the concrete (collecting) transformer $F_V^\natural$, defined as a canonical extension of the transformer F_V specified

in Def. (6.7). This approximation on functional spaces is formally described by an higher-order Galois connection [28].

Generically, given an expression of type E and some domain of interpretation L, the abstract semantics $F^\sharp : E \times L^\sharp \overset{m}{\longmapsto} L^\sharp$ is induced from the concrete semantics by calculus. The calculation process consists, for any given set approximation described by a Galois connection $\langle L^\natural, \subseteq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle L^\sharp, \sqsubseteq\rangle$, in applying the functional abstraction to the semantic transformers:

$$\langle L^\natural \overset{m}{\longmapsto} L^\natural, \dot{\subseteq}\rangle \underset{\alpha^\triangleright}{\overset{\gamma^\triangleright}{\leftrightarrows}} \langle L^\sharp \overset{m}{\longmapsto} L^\sharp, \dot{\sqsubseteq}\rangle \tag{6.16}$$

where

$$\begin{aligned} \alpha^\triangleright(F^\natural) &\triangleq \alpha \circ F^\natural \circ \gamma \\ \gamma^\triangleright(F^\sharp) &\triangleq \gamma \circ F^\sharp \circ \alpha \end{aligned} \tag{6.17}$$

The point-wise orderings $\dot{\subseteq}$ and $\dot{\sqsubseteq}$ are defined over the domains of $F^\natural$ and $F^\sharp$, respectively. Then, from the soundness requirement [28, Section 8.1] and from the fixpoint abstraction described in Section 3.7, it follows that $F^\sharp$ is an overapproximation such that:

$$F^\sharp[\![E]\!] \;\dot{\sqsupseteq}\; \alpha^\triangleright(F^\natural[\![E]\!]) \tag{6.18}$$

The main advantage of the calculational approach is that the formal specification obtained by calculus can be easily transformed into declarative code, in particular Haskell declarative code. The objective of the calculational design is to illustrate the constructive aspect of the approach of abstract interpretation, which avoid the verification *a posteriori* by means of a soundness relation using the concretization function only [23]. In fact, we address not only the soundness problem but also the precision problem by including the abstraction function in the calculation process.

For the example of the instruction **Add** introduced before, we proceed with the induction of the abstract semantic transformer using Def. (6.18). Inspecting the concrete Haskell semantic transformer, f_V, (Section 6.6.2), one can see that the values of specific registers, *reg1* and *reg2*, need to be accessed from the input register environment r. By definition, given a register name $n \in \mathbb{N}$, its concrete value in the environment $\rho \in R$ is accessed by $\rho(n) \in \mathbb{W}$. In Haskell, the environment function ρ is defined by the function *getReg*.

The environment value $\rho(n)$ is obtained by interpreting the operators $\langle Op\rangle$, in particular register operands **Reg** $\langle RegisterName\rangle$, as defined in the BNF specification of the ARM instruction set in Fig. 6.2. Accordingly, the definition of the abstract register environment $\rho^\sharp \in R^\sharp$ can be obtained by calculus using (6.18) when a Galois connection exists between the concrete and the abstract register domains. So far, we have defined such Galois connection in (6.15), but only for register names between **R0** and **R10**.

Notwithstanding, we prove that, by using the algebraic properties of Galois connections, in particular the *lower closure operator* $\alpha \circ \gamma \,\dot{\sqsubseteq}\, id$, it is possible to induce the abstract version

of the map function $\rho(n)$, when α and γ form a pair of adjoined functions defined for all $\mathbb{N}_{\mathbb{V}}$. The induction of the abstract map function $\rho^\sharp$ is performed at denotation level by providing an abstract interpretation of syntactic phrases like (**Reg** *reg*). By design, we assume that register abstract arithmetics are always performed using intervals. Therefore, we instantiate the generic pair of adjoined functions $\langle \alpha, \gamma \rangle$ of Def. (6.17), as the pair of adjoined function $\langle \dot{\alpha}, \dot{\gamma} \rangle$, defined in (6.15).

For any register $n \in \mathbb{N}_{\mathbb{V}}$ and for for an abstract environment $\rho^\sharp \in R^\sharp_\nu : \rho^\sharp \neq \bot^\sharp_\nu$, the abstract interpretation $[\![\textbf{Reg}\ reg]\!]^\sharp$ is obtained by calculus as:

$$
\begin{aligned}
& \overset{\triangleright}{\alpha}([\![\textbf{Reg}\ reg]\!])\,\rho^\sharp \\
= \quad & \wr \text{ The collecting semantics is the canonical extension of standard int. } \int \\
& \dot{\alpha}(\{\rho \mid \exists \rho \in R^\natural : [\![\textbf{Reg}\ reg]\!]\ \rho = \rho(reg)\}) \\
= \quad & \wr \text{ Haskell definition of } \rho(reg) \text{ and definition of } \dot{\alpha} \int \\
& \alpha\ (\{getReg\ r\ reg \mid r \in \dot{\gamma}(\rho^\sharp)\}) \\
\sqsubseteq \quad & \wr \text{ precedence of } \alpha,\ getReg^\sharp \text{ is the abstract version of } getReg \int \\
& getReg^\sharp\ (\dot{\alpha}\ \{r \mid r \in \dot{\gamma}(\rho^\sharp)\})\ reg \\
\sqsubseteq \quad & \wr\ \dot{\gamma} \text{ introduces no loss of information: } \forall p \in \dot{\alpha}(\dot{\gamma}(p)) \sqsubseteq^\sharp_\nu p \int \\
& getReg^\sharp\ r^\sharp\ reg \\
= \ & \rho^\sharp(reg)
\end{aligned}
$$

Hence, the abstract register set $R^\sharp$ is defined as a map from register names to the abstract values defined in the Haskell co-product type **RegVal**. The same way as the concrete map, the access functions to abstract environment are $setReg^\sharp$ and $getReg^\sharp$.

$$
\begin{aligned}
& setReg^\sharp :: R^\sharp \to [(\textbf{RegisterName}, \textbf{RegVal})] \to R^\sharp \\
& setReg^\sharp = (\ //\ ) \\
& getReg^\sharp :: R^\sharp \to \textbf{RegisterName} \to \textbf{RegVal} \\
& getReg^\sharp = (\ !\ )
\end{aligned}
$$

Next, we proceed to the calculation of the abstract instruction semantics transformer $F^\sharp_V$ for the same example instruction **Add** introduced in Section 6.6.2. For future reference, the operation $(+^\sharp)$ denotes the sum operator $(+)$ in the interval domain.

As explained in Section 6.3 in Fig. 6.3, in Load/Store architectures like the ARM9, the arithmetic operations performed inside the "ALU" functional unit during the *Execution* pipeline stage only allow registers as operands. Therefore, we conclude that the data memory is not affected during the execution of **Add**. On the other hand, during the *Memory* pipeline state, the instruction semantics of Load and Store instructions depend both on the register file and the data memory. In this way, we can define two different instruction semantics transformers,: the first is designated by F_R, and is used during the *Execution* pipeline stage; the second is designated by F_D, and is used during the *Memory* pipeline stage.

$$F_R \in Instr \mapsto R \mapsto R$$
$$F_D \in Instr \mapsto D \mapsto D$$

The abstract interpreter $F_R^\sharp$ is then obtained by calculus, considering an initial abstract environment $\rho^\sharp \in R_\nu^\sharp : \rho^\sharp \neq \bot_\nu^\sharp$:

$$\begin{aligned}
& \dot{\alpha}(F_R^\natural [\![\textbf{Add}\ (\textbf{Reg}\ reg1)\ (\textbf{Reg}\ reg2)\ (\textbf{Reg}\ reg3)]\!])\ \rho^\sharp \\
= \quad & \wr\ F_R^\natural \text{ is the canonical extension of } F_R\ \int \\
& \dot{\alpha}(\{v \mid \exists r \in R^\natural : F_R\ [\![\textbf{Add}\ (\textbf{Reg}\ reg1)\ (\textbf{Reg}\ reg2)\ (\textbf{Reg}\ reg3)]\!]\ r = v\}) \\
= \quad & \wr \text{ Standard interpretation of } \textbf{Add} \text{ as defined by the Haskell function } f_V \text{ on } R\ \int \\
& \dot{\alpha}\ (\{setReg\ r\ reg1\ (v_3 + v_2) \mid r \in \dot{\gamma}(\rho^\sharp) \wedge v_2 = [\![\textbf{Reg}\ reg2]\!]\ r \wedge v_3 = [\![\textbf{Reg}\ reg3]\!]\ r\} \\
\sqsubseteq \quad & \wr\ \dot{\gamma} \text{ monotone and defining } \mathcal{U}_R = setReg\ r\ reg1\ (v_3 + v_2) \\
& \dot{\alpha}\ (\{\mathcal{U}_R \mid \exists r_2 \in \dot{\gamma}(\rho^\sharp) : v_2 = [\![\textbf{Reg}\ reg2]\!]\ r_2 \wedge \exists r_3 \in \dot{\gamma}(\rho^\sharp) : v_3 = [\![\textbf{Reg}\ reg3]\!]\ r_3\}) \\
\sqsubseteq \quad & \wr\ \gamma \circ \alpha \text{ is extensive } (\gamma \circ \alpha \sqsupseteq id),\ \alpha \text{ is monotone}\ \int \\
& \dot{\alpha}\ (\mathcal{U}_R \mid v_2 \in \gamma \circ \alpha(\{v \mid \exists r \in r^\natural : [\![\textbf{Reg}\ reg2]\!]\ r = v\}) \wedge \\
& \qquad v_3 \in \gamma \circ \alpha(\{v \mid \exists r \in r^\natural : [\![\textbf{Reg}\ reg3]\!]\ r = v\})\}) \\
\sqsubseteq \quad & \wr \text{ definition of } [\![\textbf{Reg}\ reg]\!]^\sharp,\ \gamma \text{ and } \alpha \text{ are monotone}\int \\
& \dot{\alpha}\ (\{\mathcal{U}_R \mid v_2 \in \gamma\ ([\![\textbf{Reg}\ reg2]\!]^\sharp\ \rho^\sharp) \wedge v_3 \in \gamma\ ([\![\textbf{Reg}\ reg3]\!]^\sharp\ \rho^\sharp)\}) \\
\sqsubseteq \quad & \wr \text{ Def. (6.12) of } \gamma\ ,\ i_2 = [\![\textbf{Reg}\ reg2]\!]^\sharp\ \rho^\sharp, \text{ and } i_3 = [\![\textbf{Reg}\ reg3]\!]^\sharp\ \rho^\sharp\int \\
& \dot{\alpha}\ (setReg\ r\ reg1\ (\{z \in \mathbb{W} \mid l_2 \leqslant z \leqslant u_2\}\ +^\sharp\ \{z \in \mathbb{W} \mid l_3 \leqslant z \leqslant u_3\}) \\
& \qquad \mid l_2 = min(\gamma(i_2)) \wedge u_2 = max(\gamma(i_2)) \wedge \\
& \qquad\ \ l_3 = min(\gamma(i_3)) \wedge u_3 = max(\gamma(i_3))) \\
\sqsubseteq \quad & \wr \text{ precedence of } \dot{\alpha}, \text{ Def. (6.11) of } \alpha,\ setReg^\sharp \text{ is the abstraction of } setReg\ \int \\
& (setReg^\sharp\ r^\sharp\ reg1\ (\alpha(\{z \in \mathbb{W} \mid l_2 \leqslant z \leqslant u_2\})\ +^\sharp\ \alpha(\{z \in \mathbb{W} \mid l_3 \leqslant z \leqslant u_3\})) \\
\sqsubseteq \quad & \wr \text{ Def. (6.12) of } \gamma\int \\
& setReg^\sharp\ r^\sharp\ reg1\ (i_3\ +^\sharp\ i_2)
\end{aligned}$$

The induced abstract interpreter $f_R^\sharp$ is directly transformed into Haskell declarative code. The abstract definition of $+^\sharp$ in Haskell is obtained by an instantiation of the type class $(Num\ a)$ and the overloading of the operator $(+)$. For convenience, the interval arithmetics is performed on integer values. In this way, we convert values of type **Word32** into values of type **Integer**, and vice-versa, using the functions *toInt32* and *fromIntegral*, respectively.

```
instance Num Interval where
  (a, b) + (c, d) = let normalize (a, b) = if a ⩽ b then (a, b) else (b, a)
                        (a', b') = (toInt32 a, toInt32 b)
                        (c', d') = (toInt32 c, toInt32 d)
                        (x, y)   = normalize (a' + c', b' + d')
                    in (fromIntegral x :: Word32, fromIntegral y :: Word32)
```

Taking advantage of the parametric polymorphism of the type class ($Num\ a$), the interval arithmetics can be easily extended to the inductive datatype **RegVal**.

```
instance Num RegVal where
  AbstVal a + AbstVal b  = AbstVal (a + b)
  ConcVal a + ConcVal b = ConcVal (a + b)
```

Finally, the abstract instruction semantics for the instruction **Add** with two register operands is defined in Haskell by the function $f_R^\sharp$. Comparatively to the concrete instruction semantics given in Section 6.3.1, the only syntactic difference resides in the use of the abstract versions of $getReg^\sharp$ and $setReg^\sharp$, so that they can type check with the abstract register domain $R^\sharp$.

```
f_R^♯ :: Instr → R^♯ → R^♯
f_R^♯ (Add (Reg reg1) (Reg reg2) (Reg reg3)) r^♯
 = let i2 = getReg^♯ r reg2
       i3 = getReg^♯ r reg3
   in setReg^♯ r^♯ reg1 (v2 + v3)
```

6.6.4.2 Backward Abstract Interpretation of Operands

Backward abstract interpretation is required for the analysis of conditional instructions, such **Bne**, **Beq**, etc., with the objective to detect the existence of sound preconditions for the conditional instruction to be executed. More precisely, these preconditions correspond to the interval values of the specific registers used by the comparison instruction **Cmp** that is executed before any of referred conditional instructions. If the result of the backward analysis is the undefined element ($\bot_\nu^\sharp$), then the analysis of the conditional instruction is not performed.

First we define the *backward collection semantics* in terms of an *additive* semantic transformer $B^\natural$, defined as denotational interpretations for operands $op \in Op$. It defines the subset of possible "ascendant" operand values such that the interpretation of an operand produces a value belonging to a given set P [28].

$$\begin{aligned} B^\natural &\in Op \mapsto \wp(R) \overset{\text{a}}{\longmapsto} \wp(\mathbb{W}) \overset{\text{a}}{\longmapsto} \wp(R) \\ B^\natural[\![O]\!](R^\natural)\,P &\triangleq \{r \in R^\natural \mid \exists v \in P : f_R[\![O]\!]\,r = v\} \end{aligned} \tag{6.19}$$

Since operands of interest to the analysis are the register names for which the interval abstraction is defined, we restrict the backward abstract interpretation to the use of the abstract register domain $R_\nu^\sharp$, defined in Section 6.3. In the same way as for forward abstract interpretations, the abstract semantics $B_\nu^\sharp$ is obtained from the collecting semantics $B^\natural$ by calculus. Given the pair of adjoined functions $\langle \dot\alpha, \dot\gamma \rangle$, defined by the Galois connection (6.15) between the lattices $R^\natural(\subseteq)$ and $R_\nu^\sharp(\dot\sqsubseteq_\nu^\sharp)$, and the pair of adjoined functions $\langle \alpha, \gamma \rangle$, defined by the Galois connection (6.10) between the complete lattices $(\wp(\mathbb{W}))(\subseteq)$ and $(\mathbb{V}(\mathbb{W}))(\sqsubseteq_\nu^\sharp)$. For any possible approximation $\langle \dot\alpha, \dot\gamma \rangle$ we define the following monotonic functional abstraction:

$$\langle \wp(R) \overset{\text{m}}{\longmapsto} \wp(\mathbb{W}) \overset{\text{m}}{\longmapsto} \wp(R), \ddot\subseteq \rangle \underset{\alpha^\triangleleft}{\overset{\gamma^\triangleleft}{\leftrightarrows}} \langle R_\nu^\sharp \overset{\text{m}}{\longmapsto} \mathbb{V}(\mathbb{W}) \overset{\text{m}}{\longmapsto} R_\nu^\sharp, \ddot\sqsubseteq_\nu^\sharp \rangle$$

$$\begin{aligned} \alpha^{\triangleleft}(\Phi) &\triangleq \lambda r^\sharp \in R^\sharp_\nu \bullet \lambda p \in \mathbb{V}(\mathbb{W}) \bullet \dot{\alpha}(\Phi(\dot{\gamma}(r^\sharp))\,\gamma(p)) \\ \gamma^{\triangleleft}(\varphi) &\triangleq \lambda r \in \wp(R) \bullet \lambda P \in \wp(\mathbb{W}) \bullet \dot{\gamma}(\varphi(\dot{\alpha}(r))\,\alpha(P)) \end{aligned} \tag{6.20}$$

where the point-wise orderings $\ddot{\subseteq}$ and $\ddot{\sqsubseteq}^\sharp_\nu$ are defined for $B^\natural$ and $B^\sharp_\nu$, respectively:

$$\begin{aligned} \Phi \ddot{\subseteq} \Psi &\triangleq \forall r \in \wp(R) : \forall P \in \wp(\mathbb{W}) : \Phi(r)P \subseteq \Psi(r)P, \\ \varphi \ddot{\sqsubseteq}^\sharp_\nu \psi &\triangleq \forall r^\sharp \in R^\sharp_\nu : \forall p \in \mathbb{V}(\mathbb{W}) : \varphi(r^\sharp)p \sqsubseteq^\sharp_\nu \psi(r^\sharp)p \end{aligned}$$

The definition of the higher-order Galois connection $\langle \alpha^{\triangleleft}, \gamma^{\triangleleft} \rangle$ is obtained directly from the commutative diagram of Fig. 6.7. From the soundness requirement, it follows that given an

$$\begin{array}{ccccc} p \in \mathbb{V}(\mathbb{W}), & & r^\sharp \in R^\sharp_v(\sqsubseteq^\sharp_v) & \xrightarrow{\varphi} & R^\sharp_v(\sqsubseteq^\sharp_v) \\ \gamma \uparrow \downarrow \alpha & & \dot{\gamma} \uparrow \downarrow \dot{\alpha} & & \dot{\gamma} \uparrow \downarrow \dot{\alpha} \\ P \in \wp(\mathbb{W}), & & R \in R^\natural(\subseteq) & \xrightarrow{\Phi} & R^\natural(\subseteq) \end{array}$$

Figure 6.7: Commutative diagram used for backward abstract interpretation

abstract property p, an abstract environment $r^\sharp$ and an abstract partial order $\ddot{\sqsubseteq}^\sharp_\nu$, the value obtained by applying the abstract transformer φ must approximate the value that would be obtained by applying the concrete transformer Φ. The same applies for the concretization function $\gamma^{\triangleleft}$, which requires that, given a concrete property P and a concrete environment R, Φ is over-approximated by φ using $\ddot{\subseteq}$. Hence, the overapproximation $B^\sharp_v$ is such that:

$$B^\sharp_v[\![O]\!] \ddot{\sqsupseteq}^\sharp_\nu \alpha^{\triangleleft}(B^\natural[\![O]\!])$$

We are interested in the constant operands and operands that access a register value from the register map. For any abstract register environment $\rho^\sharp \in R^\sharp_\nu : \rho^\sharp(r1) \neq \bot^\sharp_\nu$ and $p \in R^\sharp_\nu$, we have:

1. when $O = (\textbf{Con}\ c) \in Op$ and $c \in \mathbb{W}$ is a constant, then:

$$\begin{aligned} & B^\sharp_v[\![\textbf{Con}\ c]\!]\,(\rho^\sharp)\,p \\ = \ & \dot{\alpha}(\{\rho \in \dot{\gamma}(\rho^\sharp) \mid \exists c \in \gamma(p) : [\![\textbf{Con}\ c]\!]\rho = c\}) \\ = \ & \quad \wr \text{Standard interpretation of } [\![\textbf{Con}\ c]\!] \int \\ & \dot{\alpha}(\{\rho \in \dot{\gamma}(\rho^\sharp) \mid c \in \gamma(p)\}) \\ = \ & \quad \wr \text{conditional notation } \textbf{if then else} \int \\ & \textbf{if } (c \in \gamma(p)) \textbf{ then } \dot{\alpha}(\dot{\gamma}(\rho^\sharp)) \textbf{ else } \dot{\alpha}(\emptyset) \\ \dot{\sqsubseteq}^\sharp_\nu \ & \quad \wr\, \dot{\alpha} \circ \dot{\gamma} \text{ is reductive } (\dot{\alpha} \circ \dot{\gamma} \sqsubseteq^\sharp_\nu id) \int \\ & \textbf{if } (c \in \gamma(p)) \textbf{ then } \rho^\sharp \textbf{ else } \bot^\sharp_\nu \end{aligned}$$

   The definition of $B^\sharp_\nu$ in Haskell is a function called *back* that uses the function *conc* to determine if the property p is an element of the list of concrete values:

$$
\begin{aligned}
&\mathit{back} :: \mathit{Op} \to R^\sharp \to \mathbf{RegVal} \to R^\sharp \\
&\mathit{back}\ (\mathbf{Con}\ c)\ r^\sharp\ p = \mathbf{if}\ \mathit{elem}\ c\ (\mathit{conc}\ p)\ \mathbf{then}\ r^\sharp\ \mathbf{else}\ \mathit{bottom}
\end{aligned}
$$

2. when $O = (\mathbf{Reg}\ r1) \in Op$ and $r1$ is a register name,

$$
\begin{aligned}
& B_v^\sharp[\![\mathbf{Reg}\ r1]\!]\,(\rho^\sharp)\,p \\
=\ & \dot{\alpha}(\{\rho \in \dot{\gamma}(\rho^\sharp) \mid \exists \rho(r1) \in \gamma(p) : [\![\mathbf{Reg}\ r1]\!]\rho = \rho(r1)\}) \\
=\ & \quad \wr \text{ Standard interpretation of } [\![\mathbf{Reg}\ r1]\!] \int \\
& \dot{\alpha}(\{\rho \in \dot{\gamma}(\rho^\sharp) \mid \mathit{getReg}\ r\ r1 \in \gamma(p)\}) \\
=\ & \quad \wr \text{ def. of } \dot{\gamma} \int \\
& \dot{\alpha}(\{\rho \mid \forall r2 \neq r1 : \mathit{getReg}\ r\ r2 \in \gamma\,(\mathit{getReg}^\sharp\ r^\sharp\ r2)\ \wedge \\
& \qquad\qquad \mathit{getReg}\ r\ r1 \in \gamma\,(\mathit{getReg}^\sharp\ r^\sharp\ r1)\ \cap\ \gamma(p)\}) \\
=\ & \quad \wr\ \gamma \text{ is a complete morphism} \int \\
& \dot{\alpha}(\{\rho \mid \mathit{getReg}\ r\ r2 \in \gamma\,(\mathit{getReg}^\sharp\ r^\sharp\ r2)\ \wedge \\
& \qquad \mathit{getReg}\ r\ r1 \in \gamma\,(\mathit{getReg}^\sharp\ r^\sharp\ r1 \sqcap p))\}) \\
=\ & \quad \wr \text{ let notation and definition of } \mathit{back} \int \\
& \mathbf{let}\ \mathit{back} = \lambda \mathit{reg} \to \mathbf{if}\ \mathit{reg} = r1 \\
& \qquad\qquad\qquad\qquad \mathbf{then}\ \mathit{setReg}^\sharp\ r^\sharp\ r1\ ((\mathit{getReg}\ r\ r1) \sqcap p) \\
& \qquad\qquad\qquad\qquad \mathbf{else}\ r^\sharp \\
& \mathbf{in}\ \dot{\alpha}(\{\rho \mid \mathit{getReg}\ r\ r2 \in \gamma\,(\mathit{back}\ r2)\ \ \wedge \mathit{getReg}\ r\ r1 \in \gamma\,(\mathit{back}\ r1)\}) \\
=\ & \quad \wr \text{ definition of } \dot{\gamma} \int \\
& \dot{\alpha}(\{\rho \mid \rho\ \ \in \dot{\gamma}\ \mathit{back}\}) \\
\dot{\sqsubseteq}_\nu^\sharp\ & \quad \wr\ \dot{\alpha} \circ \dot{\gamma} \text{ is reductive and } r2 \text{ cannot occur in } \mathit{Reg}\ r1 \int \\
& \mathit{setReg}^\sharp\ r^\sharp\ r1\ ((\mathit{getReg}^\sharp\ r^\sharp\ r1) \sqcap p)
\end{aligned}
$$

In this way, the Haskell definition follows directly from the calculation process by associating the function *meet* to the greatest lower bound operator ($\sqcap_\nu^\sharp$).

$$
\begin{aligned}
&\mathit{back} :: \mathit{Op} \to R^\sharp \to \mathbf{RegVal} \to R^\sharp \\
&\mathit{back}\ (\mathbf{Reg}\ r1)\ r^\sharp\ p = \mathit{setReg}^\sharp\ r^\sharp\ r1\ \$\ (\mathit{getReg}^\sharp\ r^\sharp\ r1)\ `\mathit{meet}`\ p
\end{aligned}
$$

6.6.4.3 Forward Abstract Interpretation of the 'Cmp' instruction

The backward abstract interpretation of expressions with abstract syntax $\langle Op \rangle$ is required for the forward abstract interpretation of the comparison instruction '**Cmp**', because it determines which segments of an abstract register environment will cause future interpretations of conditional instructions to branch the "program counter". Moreover, since any conditional instruction, such as '**Bne**', '**Beq**', etc., can be executed after the comparison instruction '**Cmp**', the referred interval segments must be computed for any elementary condition, in particular, $(<)$, $(==)$, or $(>)$.

We modify the treatment of conditional expressions presented by Cousot in [28] in conformity with the interval semantics of Kindahl in [73]. The reason for this is that the abstract interpretation for conditional expressions in [28] are generic, i.e. independent from the abstract domain. When using the interval abstraction, the precision of the static analysis can be improved by applying the intricate interval semantics in [73].

The intuition behind the backward abstract interpretation of boolean expressions on intervals is that conditions behave as filters which interpretation is restricted to a given interval. Let $v_1 \underline{c} v_2$ be a boolean expression using the condition $\underline{c}$. First, we evaluate the segments of the interval values v_1 and v_2 which satisfy the boolean condition. Then, we restrict the evaluation of right hand side, v_1, using an extension of the previously computed segment for v_2, and vice-versa. More precisely, this extension is defined by the widening operators $\triangledown^1$ and $\triangledown^2$ defined in Table (6.4), which depend on the conditional operation $\underline{c}$ being evaluated. The functions *above* and *below* take an interval as argument and return a second interval according to the following definition:

$$\begin{aligned} above\,([l,u]) &\triangleq [l,+\infty] \\ below\,([l,u]) &\triangleq [-\infty,u] \end{aligned}$$

Table 6.4: Widening intervals for boolean expressions

Operation	$\triangledown^1$	$\triangledown^2$
$v_1 < v_2$	$below\,(v_2) \ominus 1$	$above\,(v_1) \oplus 1$
$v_1 > v_2$	$above\,(v_2) \oplus 1$	$above\,(v_1) \ominus 1$
$v_1 == v_2$	v_1	v_2

In the intricate interval semantics the comparison $\underline{c}$ is checked for each pairs of values $\langle v_1, v_2\rangle$. Hence, we require the definition of pair of adjoined functions $\langle \alpha^2, \gamma^2\rangle$ to obtain the non-relational/componentwise abstraction of properties of pairs of values:

$$\alpha^2(P) \triangleq \langle \alpha(\{v_1 \mid \exists v_2 : \langle v_1, v_2\rangle \in P\}), \alpha(\{v_2 \mid \exists v_1 : \langle v_1, v_2\rangle \in P\})\rangle, \tag{6.21}$$

$$\gamma^2(\langle p_1, p_2\rangle) \triangleq \{\langle v_1, v_2\rangle \mid v_1 \in \gamma(p_1) \wedge v_2 \in \gamma(p_2)\} \tag{6.22}$$

with the componentwise ordering $\sqsubseteq^2$:

$$\langle p_1, p_2\rangle \sqsubseteq^2 \langle q_1, q_2\rangle \triangleq p_1 \sqsubseteq q_1 \wedge p_2 \sqsubseteq q_2 \tag{6.23}$$

As mentioned in Section 6.6.4.1, the forward abstract interpretation of an instruction executed by the "ALU" functional unit is obtained by calculus from the corresponding concrete semantics F_R. In the case of the comparison instruction '**Cmp**', the concrete semantics is given by the denotational interpretation f_R written in Haskell. The functions *cpsrSetN*, *cpsrSetZ* and *cpsrSetC* update the program *status* register (**CPSR**) by setting the respective 1-bit condition flags, "Negative", "Zero", "Carry".

$$
\begin{aligned}
&f_R\ (\mathbf{Cmp}\ (\mathbf{Reg}\ reg1)\ (\mathbf{Reg}\ reg2))\ regs\\
&\quad = \mathbf{let}\ val1 = getReg\ regs\ reg1\\
&\qquad\quad val2 = getReg\ regs\ reg2\\
&\qquad\quad regs' = setReg\ \mathbf{CPSR}\ (fromIntegral\ 0 :: \mathbf{Word32})\ regs\\
&\qquad \mathbf{in\ if}\ val1 < val2\\
&\qquad\qquad \mathbf{then}\ cpsrSetN\ regs\\
&\qquad\qquad \mathbf{else\ if}\ val1 \equiv val2\\
&\qquad\qquad\qquad \mathbf{then}\ cpsrSetZ\ regs\\
&\qquad\qquad\qquad \mathbf{else}\ cpsrSetC\ regs
\end{aligned}
$$

Before describing the calculation of $f_R^\sharp$, we have to re-define the type of the **CPSR** register. In Section 6.6.3, the abstract value of the *status* register was defined using the datatype **Word32**. However, in order to perform abstract interpretations of conditional instructions, the computation of interval segments is required for each of the elementary conditions, $(<)$, $(==)$ and $(>)$. Additionally, interval segments with elementary abstract information are required to compute the "intricate" interval segments for conditions that are expressed as combinations of the elementary conditions, e.g. $(\geqslant)$.

For this purpose, a new Haskell datatype called **Control** was defined to include not only the existent 32-bit "status" word, but also the results of the backward interpretation of the operands of the instruction '**Cmp**'. Using the record syntax of Haskell, the segment corresponding to the condition $(<)$ is accessed using the function *lessThan*, the segment corresponding to the condition $(==)$ is accessed using the function *equals*, and the segment corresponding to the condition $(>)$ is accessed using the function *greaterThan*. As opposed to the *concrete* "status" register, additional control bits are used to implement a path-sensitive data flow analysis which are set when in the presence of alternative paths or infeasible paths.

Finally, the record function *segments* is only used during abstract interpretations of conditional instructionss, for example, during the analysis of elementary branch instructions **Blt**, **Beq**, **Bgt**, or the "intrincate" branch instructions, e.g. **Bge**. By the fact that, according to a weak topological order, at the head of a loop is always a conditional instruction, the value returned by the function *segments* is used to detect fixpoint stabilization of the *value analysis* inside a loop.

Since the chaotic fixpoint strategy mimics execution order, if the function *segments* returns **Bottom**, the preconditions to analysis the loop do not hold. Then, the analysis proceeds on the fall-through path of that loop. On the hand, if two consecutive fixpoint iterations over the meta-program of the loop produce the same value of *segments* at the head of the loop, then we conclude that the fixpoint condition has been reached for that particular loop.

$$
\begin{aligned}
&\mathbf{data\ Control} = \mathbf{Control}\ \{control :: \mathbf{Word32}, lessThan :: R^\sharp, equals :: R^\sharp, greaterThan :: R^\sharp,\\
&\qquad\qquad segments :: R^\sharp\}
\end{aligned}
$$

Accordingly, the coproduct defined by **RegVal** now includes an explicit constructor, **CtrlVal**, for abstract values of type **Control**. In this way, the register name **CPSR** is now included

in the set $\mathbb{N}_{\mathbb{V}}$ of register names storing a product of interval abstractions.

data RegVal = AbstVal *Interval* **| ConcVal Word32 | CtrlVal Control | Bottom**

The induction of the abstract semantic transformer $F^{\sharp}_{R}$, defined as the abstract denotational interpretation of the the instruction '**Cmp**', is formally obtained by calculus using the soundness relation of Def. (6.18). The calculation process is generic through the definition of the condition $\underline{c}$, a syntactical meta-variable for expressing the elementary conditions. The objective of the calculation is to assign the correct values to each of the elementary interval segments.

For any abstract environment $\rho^{\sharp} \in R^{\sharp}_{\nu} : \rho^{\sharp}(r1) \neq \bot^{\sharp}_{\nu} \wedge \rho^{\sharp}(r2) \neq \bot^{\sharp}_{\nu}$, we have:

$$
\begin{aligned}
& \dot{\alpha}^{\triangleright}(F^{\natural}_{R}[\![\mathbf{Cmp}(\mathbf{Reg}\ r1)(\mathbf{Reg}\ r2)]\!])\,\rho^{\sharp} \\
= \ & \dot{\alpha}(\{\rho \in \dot{\gamma}(\rho^{\sharp}) \mid F_{R}[\![\mathbf{Cmp}(\mathbf{Reg}\ r1)(\mathbf{Reg}\ r2)]\!]\ \rho = \rho'\}) \\
= \ & \quad \wr \text{ Standard interpretation of } F_R \int \\
= \ & \dot{\alpha}(\{\rho \in \dot{\gamma}(\rho^{\sharp}) \mid \exists v_1, v_2 \in \mathbb{W} : v_1 = [\![\texttt{Reg r1}]\!]\ \rho\ \wedge\ v_2 = [\![\texttt{Reg r2}]\!]\ \rho\ \wedge \\
& \qquad\qquad (v_1 < v_2 \vee v_1 == v_2 \vee v_1 > v_2)\}) \\
= \ & \quad \wr\ \dot{\gamma} \circ \dot{\alpha} \text{ is extensive } (\dot{\gamma} \circ \dot{\alpha} \sqsupseteq id) \text{ and definging } a\ \underline{c}\ b \triangleq a < b \mid a == b \mid a > b \int \\
& \dot{\alpha}(\{\rho \in \dot{\gamma}(\rho^{\sharp}) \mid \exists v_1 \in \gamma(\dot{\alpha}(\{v_1 \mid \exists \rho \in \dot{\gamma}(\rho^{\sharp}) : [\![\mathbf{Reg}\ r1]\!]\ \rho = v_1\})) : \\
& \qquad\qquad \exists v_2 \in \gamma(\dot{\alpha}(\{v_2 \mid \exists \rho\ \in \dot{\gamma}(\rho^{\sharp}) : [\![\mathbf{Reg}\ r2]\!]\ \rho = v_2\})) : \\
& \qquad\qquad v_1 = [\![\mathbf{Reg}\ r1]\!]\ \rho\ \wedge\ v_2 = [\![\mathbf{Reg}\ r2]\!]\ \rho\ \wedge\ v_1\ \underline{c}\ v_2\}) \\
= \ & \quad \wr \text{ definition of the Galois connection } \langle \dot{\alpha}, \dot{\gamma} \rangle \int \\
& \dot{\alpha}(\{\rho \in \dot{\gamma}(\rho^{\sharp}) \mid \exists v_1 \in \gamma([\![\mathbf{Reg}\ r1]\!]^{\sharp}\ \rho^{\sharp}) : \exists v_2 \in \gamma([\![\mathbf{Reg}\ r2]\!]^{\sharp}\ \rho^{\sharp}) : \\
& \qquad\qquad v_1 = [\![\mathbf{Reg}\ r1]\!]\ \rho\ \wedge v_2 = [\![\mathbf{Reg}\ r2]\!]\ r\ \wedge v_1\ \underline{c}\ v_2\}) \\
= \ & \quad \wr \text{ let notation} \int \\
& \mathbf{let}\ \langle p_1, p_2 \rangle = \langle [\![\mathbf{Reg}\ r1]\!]^{\sharp}\ \rho^{\sharp}, [\![\mathbf{Reg}\ r2]\!]^{\sharp}\ r^{\sharp} \rangle \\
& \mathbf{in}\ \dot{\alpha}(\{\rho \in \dot{\gamma}(\rho^{\sharp}) \mid \exists v_1 \in \gamma(p_1) : \exists v_2 \in \gamma(p_2) : \\
& \qquad\qquad v_1 = [\![\mathbf{Reg}\ r1]\!]\ \rho\ \wedge\ v_2 = [\![\mathbf{Reg}\ r2]\!]\ \rho\ \wedge\ v_1\ \underline{c}\ v_2\}) \\
= \ & \quad \wr \text{ set theory define a relation between the sets } \gamma(p_1) \text{ and } \gamma(p_2) \int \\
& \mathbf{let}\ \langle p_1, p_2 \rangle = \langle [\![\mathbf{Reg}\ r1]\!]^{\sharp}\ \rho^{\sharp}, [\![\mathbf{Reg}\ r2]\!]^{\sharp}\ \rho^{\sharp} \rangle \\
& \mathbf{in}\ \dot{\alpha}(\{\rho \in \dot{\gamma}(\rho^{\sharp}) \mid \exists \langle i_1, i_2 \rangle \in \{\langle i'_1, i'_2 \rangle \mid i'_1 \in \gamma(p_1) \wedge i'_2 \in \gamma(p_2)\ \wedge\ v_1\ \underline{c}\ v_2\} : \\
& \qquad\qquad v_1 = [\![\mathbf{Reg}\ r1]\!]\ \rho\ \wedge\ v_2 = [\![\mathbf{Reg}\ r2]\!]\ \rho\}) \\
\dot{\sqsubseteq}^{\sharp}_{\nu} \ & \quad \wr\ \gamma^2 \circ \alpha^2 \text{ is extensive and } \dot{\alpha} \text{ is monotone} \int \\
& \mathbf{let}\ \langle p_1, p_2 \rangle = \langle [\![\mathbf{Reg}\ r1]\!]^{\sharp}\ \rho^{\sharp}, [\![\mathbf{Reg}\ r2]\!]^{\sharp}\ \rho^{\sharp} \rangle \\
& \mathbf{in}\ \dot{\alpha}(\{\rho \in \dot{\gamma}(\rho^{\sharp}) \mid \exists \langle i_1, i_2 \rangle \in \gamma^2(\alpha^2(\{\langle i'_1, i'_2 \rangle \mid i'_1 \in \gamma(p_1) \wedge i'_2 \in \gamma(p_2)\ \wedge\ v_1\ \underline{c}\ v_2\})) : \\
& \qquad\qquad v_1 = [\![\mathbf{Reg}\ r1]\!]\ \rho\ \wedge\ v_2 = [\![\mathbf{Reg}\ r2]\!]\ \rho\}) \\
\dot{\sqsubseteq}^{\sharp}_{\nu} \ & \quad \wr \text{ defining } \check{c}(p_1, p_2, \underline{c}) \sqsupseteq^2 \alpha^2(\{\langle i'_1, i'_2 \rangle \mid i'_1 \in \gamma(p_1) \wedge i'_2 \in \gamma(p_2)\ \wedge\ v_1\ \underline{c}\ v_2\}) \int \\
& \mathbf{let}\ \langle p_1, p_2 \rangle = \langle [\![\mathbf{Reg}\ r1]\!]^{\sharp}\ \rho^{\sharp}, [\![\mathbf{Reg}\ r2]\!]^{\sharp}\ \rho^{\sharp} \rangle \\
& \mathbf{in}\ \dot{\alpha}(\{\rho \in \dot{\gamma}(\rho^{\sharp}) \mid \exists \langle i_1, i_2 \rangle \in \gamma^2(\check{c}\,(p_1, p_2, \underline{c})) : v_1 = [\![\mathbf{Reg}\ r1]\!]\ \rho\ \wedge\ v_2 = [\![\mathbf{Reg}\ r2]\!]\ \rho\})
\end{aligned}
$$

$$
\begin{aligned}
&= \quad \lbag \text{re-definition of } \langle p_1, p_2\rangle \text{ and definition of } \textit{below} \text{ and } \textit{above} \text{ widening operators} \rbag \\
&\quad \textbf{let } \langle p_1, p_2\rangle = \check{c}\,([\![\textbf{Reg } r1]\!]^\sharp\ \rho^\sharp, [\![\textbf{Reg } r2]\!]^\sharp\ \rho^\sharp, \underline{c}) \\
&\quad \textbf{in } \dot{\alpha}(\{\rho \in \dot{\gamma}(\rho^\sharp) \mid \exists i_1 \in \gamma(\triangledown^1 p_2) : [\![\textbf{Reg } r1]\!]^\sharp\ \rho^\sharp = i_1\} \cap \\
&\qquad \{\rho \in \dot{\gamma}(\rho^\sharp) \mid \exists i_2 \in \gamma(\triangledown^2 p_1) : [\![\textbf{Reg } r2]\!]^\sharp\ \rho^\sharp = i_2\}) \\
&= \quad \lbag \dot{\alpha} \text{ is a complete join morphism by definition} \rbag \\
&\quad \textbf{let } \langle p_1, p_2\rangle = \check{c}\,([\![\textbf{Reg } r1]\!]^\sharp\ \rho^\sharp, [\![\textbf{Reg } r2]\!]^\sharp\ \rho^\sharp, \underline{c}) \\
&\quad \textbf{in } \dot{\alpha}(\{\rho \in \dot{\gamma}(\rho^\sharp) \mid \exists i_1 \in \gamma(\triangledown^1 p_2) : [\![\textbf{Reg } r1]\!]^\sharp\ \rho^\sharp = i_1\}) \mathbin{\dot{\sqcap}} \\
&\qquad \dot{\alpha}(\{\rho \in \dot{\gamma}(\rho^\sharp) \mid \exists i_2 \in \gamma(\triangledown^2 p_1) : [\![\textbf{Reg } r2]\!]^\sharp\ \rho^\sharp = i_2\}) \\
\dot{\sqsubseteq}^\sharp_\nu \quad &\lbag \text{def. of backward abstract interpretation } B^\sharp_v \rbag \\
&\quad \textbf{let } \langle p_1, p_2\rangle = \check{c}\,([\![\textbf{Reg } r1]\!]^\sharp\ \rho^\sharp, [\![\textbf{Reg } r2]\!]^\sharp\ \rho^\sharp, \underline{c}) \\
&\quad \textbf{in } (B^\sharp_\nu[\![\textbf{Reg } r1]\!]\ \rho^\sharp(\gamma(\triangledown^1 p_2))) \mathbin{\dot{\sqcap}} (B^\sharp_\nu[\![\textbf{Reg } r2]\!]\ \rho^\sharp)(\gamma(\triangledown^2 p_1)))
\end{aligned}
$$

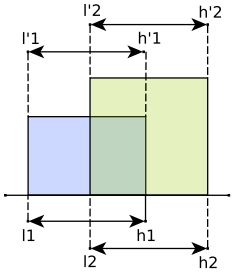
In this way, the forward abstract denotation interpretation of an instruction '**Cmp**' is given by the semantic transformer $F^\sharp_R$, parametrized by a set of conditionals $\underline{c}$ that need fully evaluated under an input abstract environment, $\rho^\sharp$. Next, we give the Haskell definition of $\check{c}$ by means of the function *pairs*. It is formally defined as $\check{c}(p_1, p_2) \sqsupseteq^2 \alpha^2(\{\langle i'_1, i'_2\rangle \mid i'_1 \in \gamma(p_1) \wedge i'_2 \in \gamma(p_2) \ \wedge\ v_1 \,\underline{c}\, v_2\})$.

Example 8. Example of segment extraction between two intervals.
Consider, as an example, that the condition $\underline{c}$ is the **LessThan** ($<$) condition. For sake of efficiency, we do not check the condition for all the possible pairs $\langle i'_1, i'_2\rangle$, but only for the lower and upper bounds of this set of pairs. Fig. 6.8 shows all the possible combinations of lower and upper bounds between two intervals $[l_1, u_1]$ and $[l_2, u_2]$, from which we can extract the abstract semantics of both intervals operands that are involved in the **LessThan** comparison.

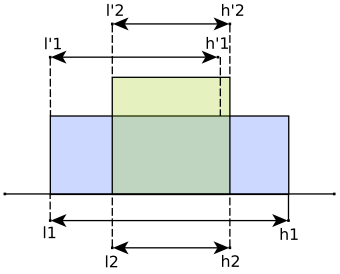
```
data Condition = LessThan | Equal | GreaterThan

pairs :: R♯ → Op → Op → Condition → (RegVal, RegVal)
pairs r♯ (Reg reg1) (Reg reg2) LessThan
  = let AbstVal (l1, h1) = getReg♯ r♯ reg1
        AbstVal (l2, h2) = getReg♯ r♯ reg2
        c = if h1 < h2 then h1
                       else h2 − 1
        b = if l1 < l2 then l2
                       else l1 + 1
    in if l1 > h2
       then (Bottom, Bottom)
       else (abst [l1, c], abst [b, h2])
```

The intervals $[l'_1, u'_1]$ and $[l'_2, u'_2]$ are the segments of the original intervals such that all values inside $[l'_1, u'_1]$ are less than all the values inside $[l'_2, u'_2]$. The function *pairs* computes these

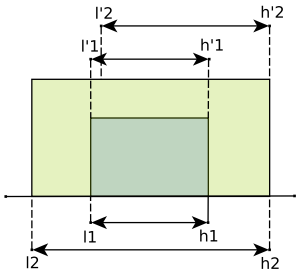
bounds and abstracts them using *abst* to obtain the effect of α^2. As mentioned, the function *pairs* has as argument of type **Condition** to specify which comparison filter is being applied.



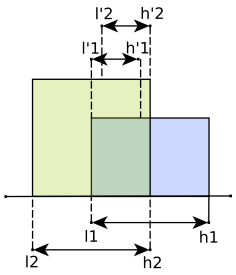
(a) Case 1



(b) Case 2



(c) Case 3



(d) Case 4

Figure 6.8: All possible cases when comparing the intervals $[l1, u1]$ and $[l2, u2]$

▲

Next, we give the definition of the function *widening*, used to obtain the pair of values $(\triangledown^1 p_2, \triangledown^2 p_1)$ that specify the intricate interval semantics. For the case where the condition is the $(<)$ condition, the definition of *widening* shows that the restriction of the first interval is the interval *below* the second interval minus one. Conversely, the restriction to the second interval is the interval *above* the first interval plus one.

$$
\begin{aligned}
&\mathit{widening} :: \mathbf{Condition} \rightarrow (\mathbf{RegVal}, \mathbf{RegVal}) \rightarrow (\mathbf{RegVal}, \mathbf{RegVal}) \\
&\mathit{widening}\ \mathbf{LessThan}\ (i1, i2) = (\mathit{below}\ i2\ -\ \mathbf{AbstVal}\ (1,1), \mathit{above}\ i1\ +\ \mathbf{AbstVal}\ (1,1)) \\
&\mathit{widening}\ \mathbf{Equal} = \mathit{id} \\
&\mathit{widening}\ \mathbf{GreaterThan}\ (i1, i2) = (\mathit{above}\ i2\ +\ \mathbf{AbstVal}\ (1,1), \mathit{below}\ i1\ -\ \mathbf{AbstVal}\ (1,1))
\end{aligned}
$$

The interval segments originated by a particular comparison condition are computed by the function *compareOps*, which arguments are the input abstract environment, the two operands of the comparison instruction and the specified condition. Its effect is an abstract environment where the two register operands are "filtered" so that the interval abstraction includes only the values for which condition holds. However, the definition of *compareOps* is an incomplete definition of the formal definition $F^{\sharp}_{R}[\![\mathbf{Cmp}\ (\mathbf{Reg}\ r1)(\mathbf{Reg} r2)]\!]\ \rho^{\sharp}$, because it considers only one particular comparison condition.

Therefore, for one particular **Condition**, its definition uses the corresponding definition of the

function *pairs*, which instantiates the pairs $\langle p_1, p_2 \rangle$, the function *widening*, which instantiates the intervals $(\nabla^1 p_2), \nabla^2 p_1)$, the function *back*, which instantiates the backward interpretation $B^\sharp_\nu$ and, finally, the function *meet*, which instantiates the greatest lower bound operator $\dot{\sqcap}^\sharp_\nu$.

$$
\begin{aligned}
&\mathit{compareOps} :: R^\sharp \to \mathit{Op} \to \mathit{Op} \to \textbf{Condition} \to R^\sharp \\
&\mathit{compareOps}\ r^\sharp\ \mathit{reg1}\ \mathit{reg2}\ c = \textbf{let}\ (w1, w2) = \mathit{widening}\ c\ \$\ \mathit{pairs}\ r^\sharp\ \mathit{reg1}\ \mathit{reg2}\ c \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{reg1}' = \mathit{back}\ \mathit{reg1}\ r^\sharp\ w1 \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{reg2}' = \mathit{back}\ \mathit{reg2}\ r^\sharp\ w2 \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in}\ \mathit{reg1}'\ `\mathit{meet}`\ \mathit{reg2}'
\end{aligned}
$$

Finally, the complete declarative definition for $F^\sharp_R[\![\textbf{Cmp}\ (\textbf{Reg}\ r1)\ (\textbf{Reg}\ r2)]\!]\ \rho^\sharp$ is given by the function $f^\sharp_r$. After comparing the input operands for all comparison conditions, the returned abstract environment contains a modified value for the register **CPSR**, which value is constructed using the results of the previous backward abstract interpretations. Using the record functions *lessThan*, *equals*, *greaterThan*, these results are assigned with the interval segments, *lt*, *eq* and *gt*, respectively. The values of the *control* word and the "intricate" *segments* are set to their undefined values (*bottom*) because they will be modified only by the branch conditional instruction that immediately follows the comparison instruction.

$$
\begin{aligned}
&f^\sharp_R\ (\textbf{Cmp}\ (\textbf{Reg}\ \mathit{reg1})\ (\textbf{Reg}\ \mathit{reg2}))\ r^\sharp \\
&\quad = \textbf{let}\ \mathit{lt} = \mathit{compareOps}\ r^\sharp\ (\textbf{Reg}\ \mathit{reg1})\ (\textbf{Reg}\ \mathit{reg2})\ \textbf{LessThan} \\
&\qquad\quad \mathit{eq} = \mathit{compareOps}\ r^\sharp\ (\textbf{Reg}\ \mathit{reg1})\ (\textbf{Reg}\ \mathit{reg2})\ \textbf{Equal} \\
&\qquad\quad \mathit{gt} = \mathit{compareOps}\ r^\sharp\ (\textbf{Reg}\ \mathit{reg1})\ (\textbf{Reg}\ \mathit{reg2})\ \textbf{GreaterThan} \\
&\qquad\quad \mathit{ctrl} = \textbf{Control}\ \{\mathit{lessThan} = \mathit{lt}, \mathit{equals} = \mathit{eq}, \mathit{greaterThan} = \mathit{gt}, \\
&\qquad\qquad\qquad\qquad \mathit{control} = \mathit{fromIntegral}\ 0 :: \textbf{Word32}, \mathit{segments} = \mathit{bottom}\} \\
&\quad\ \textbf{in}\ \mathit{setReg}^\sharp\ r^\sharp\ \textbf{CPSR}\ (\textbf{CtrlVal}\ \mathit{ctrl})
\end{aligned}
$$

6.6.5 Fixpoint Stabilization

As described in Chapter 5, the fixpoint algorithm makes use of chaotic program-specific iteration strategies determined according to the weak topological order of some program P. In the previous section, we have defined a calculational-based abstract instruction semantics for ARM9 programs using the interval abstraction. Given an ARM9 program with n instructions, the fixpoint of the program in the abstract domain is computed over the function space $F = \langle f_{(1,h)}, \ldots, f_{(i,j)}, \ldots, f_{(k,n)} \rangle$, where each data-propagation function $f_{(i,j)}$ computes a state transformation between the program labels i and j. By definition of weak topological order, the invariants map $\sigma_i = \Sigma^i_P$ is already available when the computation of the state Σ^j_P starts, implying the interpretation of sequential statements is made only once and in the right order.

The purpose of the two-level denotational meta-language is two-fold. The lower level is used to defined the type of propagation functions and the upper level is used to define combinations of propagation function according to control-flow patterns. The basic combinator is the sequential composition $(f * g)$, which uses the denotational concept of "continuation" to

specify that the function g is the continuation of f. The type signature of $(*)$ specify the type-safety of this continuation effect in the sense that f has the parametric polymorphic type $(a \to b)$ and the continuation g has the parametric polymorphic type $(b \to c)$. The advantage of the algebraic shape of the meta-language combinators is that the result type of $(*)$ is, as expected, parametric polymorphic $(a \to c)$. Given a third continuation h, the combinator $(*)$ can again be used to express the functional composition $h \circ g \circ f$. Fixpoint semantics expressed in this continuation style yield the compositional fixpoint algorithm given in Def. (5.23).

However, there are other control-flow patterns besides sequential composition, such as the recursive and pseudo-parallel combinators, and the interface adapters *split* and *merge*. Nonetheless, each of these combinators/adapters are combined between each others always by means of the sequential composition. Furthermore, the recursive combinator internally uses sequential composition to define loop unrolling as the sequence of tail calls that take the function defining the effect of the loop as the current continuation. This sequence of functional applications corresponds to Kleene sequences in the sense that only monotonic functions φ are considered, and that their least upper bounds, $lfp\ \varphi$, are obtained by applying the fixpoint operator FIX to φ, such that $FIX(\varphi) = lfp\ \varphi$. The theoretical foundations for computing fixpoints using denotational semantics are given in Chapter 2 and the Haskell definitions for the recursive operators $(\oplus)$ and $(\oslash)$ are given in Section 5.2.

Example 9. Example of the stabilization of a loop
To better understand the fixpoint algorithm, we reintroduce the our simple source code example with a '**while**' loop in Fig. 6.9(a). The ARM9 machine program is given in Fig. 6.9(b), including the labelled program points "at" the beginning and "after" each instruction. The induced chaotic iteration strategy is the following:

$$\ldots\ 6\ 10\ 11\ [\underline{12}\ 7\ 8\ 9\ 10\ 11]^*\ \ldots$$

```
int main(void) {
  int x = 3;
  while (x>0) {
    x--;
  }
  return x;
}
```
(a) Source program

```
n10: b    16              :n6
 n8: ldr  r3, [fp, #-16]  :n7  {(".L3","main")}
 n9: sub  r3, r3, #1      :n8
n10: str  r3, [fp, #-16]  :n9
n11: ldr  r3, [fp, #-16]  :n10 {(".L2","main")}
n12: cmp  r3, #0          :n11
 n7: bgt  -20             :head_12
n13: bgt  -20             :n12
```
(b) Labelled relational semantics

Figure 6.9: Source program and the corresponding labelled relational semantics

By definition of weak topological order, the instructions between the label identifiers 0 and 6 are analyzed using the sequential operator $(*)$. The same combinator is used during the analysis of the *branch-and-link* instruction '`bl 16`', which is delimited by the label identifiers 6 and 10. This puts in evidence the benefits of the relational algebra of continuations provided

by the sequential combination of state-propagation functions (once and in the right order).

Next, we give particular attention to the component of the chaotic fixpoint strategy between square parentheses ($[]^*$), and how the recursive combinator ($\oplus$) is able to compute the least fixpoint for that particular component. A fragment of the complete meta-program, i.e. the automatically derived fixpoint algorithm is:

$$\cdots \texttt{ * (cmp r3, \#0) * ((bgt -20)} \oplus \texttt{(ldr r3, [fp, \#-16]) * } \cdots \texttt{ * (sub r3, r3, \#1) * } \cdots \texttt{ * (ldr r3, [fp, \#-16]) * (cmp r3, \#0)) * (bgt -20) * } \cdots$$

Like the definition of the combinator ($*$), also the combinator ($\oplus$) is based on an algebra of binary relations, where each relation has the functional type of a continuation. The first continuation corresponds to the propagation function that iterates over the instruction '`bgt -20`', and the second continuation corresponds to the functional composition of the propagation functions that iterate of the instruction sequence '`(ldr r3, [fp, #-16])` $*\cdots*$ `(cmp r3, #0)`'. The condition to enter the loop is analyzed by iterating over the continuation of the instruction `cmp r3, #0` using, for that purpose, the backward abstract interpretation of the two instruction operands.

As described in Section 6.6.4.3, the forward abstract interpretation of the instruction '**Cmp**' uses the results of backward abstract interpretation of its two operands, as described in Section 6.6.4.2. Note that in previous meta-program, the instruction '`cmp r3, #0`' appears before entering the (sub)meta-program of the loop and also appears at the last position inside the loop. In both cases, it takes as the current continuation the propagation function that iterates over the instruction '`bgt -20`'.

Consequently, the forward abstract interpretation of the instruction '**Bgt**' must be defined. However, we do not apply the calculational method used in the previous sections because the only different between the concrete standard interpretation and the abstract interpretation resides on the representation of the register '**CPSR**'. As opposed to the 32-bit word used by a concrete register value, where the conditional flags 'Negative' (N), 'Zero' (Z), 'Carry' (C) can be set to 1 or 0, the abstract value of **CPSR** defines interval segments for each one of the condition applies.

Therefore, branch conditions are not determined in terms of the value of a control bit, which would be either 0 or 1 during run-time execution, but rather in terms of an interval segment, which is specific to a particular comparison condition that can be either the undefined element (bottom) or not. These segments are calculated by backward abstract interpretation and are essential to perform a path-sensitive data-flow analysis. This can be observed in the nondeterministic relational semantics of Fig. 6.9(b), where the instruction '`bgt -20`' appears both at the "head" and at the "exit" labels of the loop, allowing the analysis of the fall-through path of the loop. ▲

The standard (concrete) interpretation of the instruction '**Bgt**' over a register environment

is given by the Haskell semantic transformer f_R. The program counter *offset* associated to the branch instruction is provided at analysis time by the constructor **Rel**. The set of control bits 'N', 'Z' and 'C' is updated at once during the execution of the instruction '**Cmp**'. For the particular case of the instruction '**Bgt**', the condition to check is $(>)$ and is associated with the 'Carry' bit 'C', which is evaluated by means of the function *cpsrGetC*. If the bit 'C' is set to 1, then the "program counter" is updated with the provided *offset*. Otherwise, the register environment is left unaltered.

```
fR (Bgt (Rel offset)) regs
  = let pc = getReg regs R15
        pc' = if offset < 0
                 then pc − 4 − (fromIntegral (−offset))
                 else pc − 4 + (fromIntegral offset)
    in  if cpsrGetC regs ≡ 1
        then setReg regs R15 pc'
        else regs
```

The forward abstract interpretation of the instruction '**Bgt**' over an abstract register environment is given by the semantic transformer $f_R^\sharp$. Since the backward abstract interpretation uses the interval abstraction, the condition $(>)$ is evaluated by comparing the results of the function *greaterThan* with the undefined interval returned by *bottom*. If this comparison evaluates to **True**, the record function *segments* is updated with the value of *greaterThan*.

Otherwise, the complement condition $(\leqslant)$ is considered on the fall-through path by computing the least upper bound (*join*) between the results of the functions *lessThan* and *equals*. Finally, two abstract control bits are used to specify if a branch has the sufficient preconditions, i.e. an abstract value different from *bottom*, or to specify if a branch gives origin to an infeasible path. These two possibilities are modelled by an additional control bit, designated by "BranchBit", which can be set to '1' or '0' by means of the functions *setBranchBit* and *clearBranchBit*, respectively.

```
fR♯ (Bgt (Rel offset)) r♯
  = let CtrlVal status@Control {control = cpsr, greaterThan} = getReg♯ r♯ CPSR
        ConcVal pc = getReg♯ r♯ R15
        pc'' = if offset < 0
                  then pc − 4 − (fromIntegral (−offset))
                  else pc − 4 + (fromIntegral offset)
    in if greaterThan ≢ bottom
       then let ctrl = status {control = setBranchBit cpsr,
                               segments = greaterThan}
                regs' = setReg♯ r♯ R15 $ ConcVal pc''
            in setReg♯ regs' CPSR (CtrlVal ctrl)
       else let ctrl = status {control = clearBranchBit cpsr,
                               segments = join (equals status) (lessThan status)}
            in setReg♯ r♯ CPSR (CtrlVal ctrl)
```

The importance of the interval returned by the function *segments* is to detect fixpoint

stabilization of loops at their "heads". The verification of the ascending chain condition of the Kleene sequence that correspond to the loop unrolling of a recursive meta-program is performed by checking the existence of the least upper bounds of the *segments* intervals at the label identifiers of the head of the loop inside $\Sigma[\![P]\!]$. Therefore, the results of the backward abstract interpretation have to be compared after each fixpoint iteration in order to detect stabilization of this particular analysis. For this purpose, we re-define the datatype (**Env** a), first introduced in Section 5.1, and later modified in Section 6.4, so that it uses the value constructor **loopStable** to include the notion of loop stabilization.

```
data Env a = Env {value :: a, stable :: [Stable]}
data Stable = ValueStable | CacheStable | PipelineStable | LoopStable
```

Since the WCET analysis is an instantiation of the generic framework presented in Chapter 5, the effect of the function *stabilize* needs to be re-defined to reflect the stabilization of the *value analysis*. Two different cases must be considered in this process: in the first case, fixpoint stabilization is detected at every program point where the least upper bound for register and data memory abstract environments, $R^\sharp$ and $D^\sharp$, has been reached; in the second case, fixpoint stabilization is detected solely at the "heads" of loops because only the results of the backward abstract interpretation are used. In both cases, the update of the record function *stable* is performed inside the definition of the function *chaotic*, as described in Section 5.1.

As described in Section 6.4, when the stabilization in the value domain is "globally" achieved, i.e. considering all program points inside the loop, the constructor **ValueStable** is used to instrument the *program flow analysis*. As a particular case, when considering only the "head" of the loop, the constructor **LoopStable** is used to detect the stabilization of the loop unrolling when performing the *value analysis*. The overall process of fixpoint stabilization will be described next in the following way. First, we show that the polymorphic function *chaotic* must be instantiated with the abstract domain **CPU**. Then, we identify the different situations in which information about fixpoint stabilization can be updated and which functions are used accordingly. Finally, we define the instance (*Iterable* **CPU**).

```
chaotic :: (Transition r) ⇒ r → Invs CPU → CPU → Invs CPU
chaotic rel invs cpu
  = let cpu' = read invs (sink rel)
        cpu'' = join cpu' cpu
    in if cpu' ≡ bottom
         then store rel cpu invs
         else if cpu'' ≢ cpu'
                then partialStabilize rel cpu' cpu'' $ store rel cpu'' invs
                else stabilize rel invs
```

As usual, when the most recent abstract value *cpu* is available, the function *join* computes the least upper bound between this value and the value that already exists inside the invariants map at the *sink* label. If the least upper bound is equal to the previous abstract value, this means that, considering only for the "sink" program point of the input relation, the

ascending chain condition has been verified and the least fixpoint has been found. In this case, the state of the *value analysis* is updated using the function *stabilize*.

```
stabilize :: Transition r ⇒ r → Invs CPU → Invs CPU
stabilize rel = adjust (λn → n {stable = [ValueStable]}) $ (point ∘ sink) rel
```

During chaotic fixpoints where the least upper bound has not been reached for all the abstract domains contained in the composite domain denoted by **CPU**, we detect partial stabilization in the domains of interest for loop unrolling, i.e. the abstract value domains $R^\sharp$ and $D^\sharp$. If the last computed abstract value *cpu* is associated to an input-output relation which *source* label is the "head" of a loop, we apply the function *stabilizeLoop*. Otherwise, in the general case, we apply the function *stabilizeValue*.

```
partialStabilize :: (Transition r) ⇒ r → CPU → CPU → Invs CPU → Invs CPU
partialStabilize rel = if (head ∘ source) rel
                       then stabilizeLoop rel
                       else stabilizeValue rel
```

The effect of the function *stabilizeLoop* is uniquely determined by the state of the backward abstract interpretation after the forward abstract interpretation of a comparison instruction, of type '**Cmp**', followed by the forward abstract interpretation of a branch instruction, e.g. of type '**Bgt**'. At this stage, the *segments* record function of the abstract value stored inside the register **CPSR** already carries the results of a path-sensitive data-flow analysis. If such information is equal in two consecutive chaotic fixpoint iterations, then we add the data constructor **LoopStable** to the fixpoint information stored inside the *sink* label of the branch instruction.

```
stabilizeLoop :: (Transition r) ⇒ r → CPU → CPU → Invs CPU → Invs CPU
stabilizeLoop rel old new
  = if (head ∘ source) rel
      then let CtrlVal c' = getReg♯ (registers old) CPSR
               CtrlVal c'' = getReg♯ (registers new) CPSR
           in if segments c' ≡ segments c''
                then adjust (λn → n {stable = [LoopStable]}) $ (point ∘ sink) rel
                else id
      else id
```

In the general case, it is only required to detect the stabilization of the *value analysis*. For this purpose, we compare the abstract contents of the register and data memory domains, through the record function *registers* and *dataMem*, respectively. If such information is equal in two consecutive chaotic fixpoint iterations, then we add the data constructor **ValueStable** to the fixpoint information stored inside the *sink* label of the instruction.

```
stabilizeValue :: (Transition r) ⇒ r → CPU → CPU → Invs CPU → Invs CPU
stabilizeValue rel old new
  = if (registers old ≡ registers new) ∧ (dataMem old ≡ dataMem new)
      then adjust (λn → n {stable = [ValueStable]}) $ (point ∘ sink) rel
      else id
```

Finally, we instantiate the type class (*Iterable a*) for parametrized programs states (**St CPU**). The objective of the function *isFixpoint* is to determine either if there a preconditions for a loop be analyzed or if the Kleene sequence constructed during the interpretation of the intra-procedural Haskell combinator $(f + r)$ has reached the least fixpoint (here f is the continuation of the loop body and r is the continuation of the branch instruction). Hence, if the "BranchBit" stored inside the *control* of a **CPSR** register is set to 0, there are no sufficient preconditions to continue the analysis of the loop and the combinator returns the complement of the input program state with respect to the boolean condition.

Hence, if after a finite stream of tail calls $f(f(f(...)))$, the least upper bound of $FIX(f)$ has been found, the **LoopStable** constructor will be added to the list of *stable* components of an abstract **Env CPU**. In this case, the analysis of the loop terminates, giving rise to the analysis of the fall-through path.

```
instance Iterable (St CPU) where
  isFixpoint s = let (after, cert) = (label s, invs s)
                     node = cert ! (point after)
                     CtrlVal status = getReg♯ (registers (value node)) CPSR
                 in case getBranchBit (control status) of
                      0 → False
                      1 → ¬ $ elem LoopStable (stable node)
```

6.7 Cache Analysis

Modern embedded microprocessors, such as the ARM9, can execute instructions at a very high rate by employing an efficient instruction-level pipeline and a memory system that is both very large and very fast. If the size of the main memory is to sufficiently large to hold enough programs that keep the microprocessor busy, or if the access times to the main memory are too slow, the main memory will not be able to supply instructions as fast as the processor can execute them.

Unfortunately, the larger the memory is, the slower are its provided access times. The solution to this problem is the design of composite memory systems capable to combine small and fast specialized memories with a large and slow main memory. Although the technical effectiveness of this solution depends on typical program statistics, such as *spatial locality* and *temporal locality* properties of the program [106], the objective is observe the memory system as acting like a large and fast memory, at least for much of the time. This kind of behavior is provided by a specialized memory called *cache*. Consider, as an example, the Harvard architecture of ARM9 illustrated in Fig. 6.10. It comprises two different SRAM caches with separated bus links: a *data cache*, D-$, and a *instruction cache*, I-$.

Memory systems support memory hierarchy of many levels. For example, Fig. 6.10 identifies two internal cache levels: separated Level 1 (L1) caches and a common Level 2 (L2) cache.

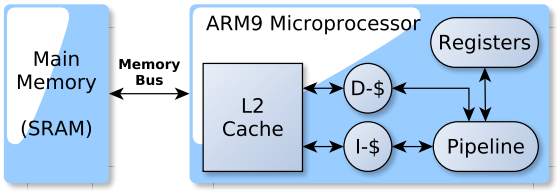


Figure 6.10: Memory hierarchy of a ARM9 microprocessor

Both L1 and L2 caches are composed of SRAM but L2 caches are much larger. In one end, external hard disk drives (HDD) are generally considered at the lowest (and the slowest) level in the memory hierarchy. At the other end, processor registers can be viewed as the elements on top of the memory hierarchy. For the complete memory hierarchy, access times typically vary from a few nanoseconds at the level of registers, tens to a few tens of nanoseconds at the levels of the internal caches, and around 100 nanoseconds at the level of the main memory.

Memory accesses that are requested by the pipeline during the "Fetch" stage are serviced directly from the instruction cache. If the requested "program counter" memory address is contained in the cache, which corresponds to a *cache hit*, then the opcode is returned at a low latency. Conversely, upon a *cache miss*, i.e. a case when the requested instruction resides in the main memory, it first needs to be transferred from the main memory to the cache, replacing existent cached data.

Data replacement inside a cache is highly relevant for WCET timing analysis. It is formally determined by a *replacement policy* and affects both temporal and spatial locality, whence the execution time of a particular program. The replacement policies are specified by the logical organization of the cache, i.e. how the cache is structured internally, and how it manages memory accesses.

In general, the main memory is logically partitioned into a set of memory blocks M of size b bytes, with the objective to reduce traffic and management overhead. Therefore, cache lines are designed to have equal size so that memory blocks can be cached as a whole. Taking advantage of spatial locality, b is usually a power of two. A memory block is then easier to find because the *block offset* is determined by most significant bits of a memory address. When an access to a memory block is issued, it is either already stored in the cache (cache hit) or it is not (cache miss).

There are caches with many different shapes and sizes, but all can be categorized in terms of a parameter called *cache associativity*. In general, caches are partitioned into *cache sets* of equal size. The size of each cache set is the associativity k of the cache. Given the total number s of equally sized cache sets, each set is identified by least significant bits of the block number, called the *index*. There are three categories for cache associativity: *direct mapped*, *k-way set associative*, and *fully associative*. The remaining most-significant bits of the address, known as the *tag*, are stored along with each cache line.

The *direct mapped* caches are also 1-*way set associative* caches. Lookup of memory blocks in this kind of caches is very simple and cheap to implement as it not necessary to compare several tags to decide, whether and where a memory block is cached within a set. On the other hand, *fully associative* caches are also *n-way set associative* caches. These caches are typically implemented using translation lookaside buffers and no *index* is needed since a memory block can be cached anywhere inside the n positions of a single cache set. In order to determine whether the memory block is currently stored in the cache, it is necessary to compare the *tag* of each memory address. Finally, a *k-way set associative* cache is a generic cache that requires both the *tag* and the *index* to lookup a cache block. For a given target latency, the hardware complexity of a generic cache ranges from the cheap solutions of *direct mapped* caches to the expensive implementations of *fully associative* caches.

The replacement policy completes the specification of a cache [8]. By the fact that the number s of cache sets is typically small, a cache set eventually fills up after a certain number of cache misses. Upon the next cache miss, the tasks of the replacement policy is to decide on the basis of a particular strategy which is the "least useful" memory block inside the cache set and to replace it inside the cache set in such a way that the number of cache misses is minimized across the whole program execution. Ideally, if it was possible to have perfect knowledge about future requests to memory blocks, an optimal replacement (OPT) algorithm would replace the memory block whose request is the farthest away in the future among all the cache memory blocks presently in the set [18]. Since the implementation of such algorithm is infeasible, the following alternative strategies are the most popular.

Prominent replacement policies are: (1) Random [146], used for example in ARM Cortex-A8; (2) LRU (Least Recently Used) [3], used for example in Intel Pentium I and MIPS 24K/34K; (3) Round-robin (or FIFO – First-In-First-Out), used for example used in Motorola PowerPC 56x, Intel XScale, ARM9, ARM11; (4) PLRU (Pseudo LRU), used for example in used in Intel Pentium II-IV and PowerPC 75x; (5) Most Recently Used (MRU) [8], used for example in Intel Nehalem; (5) Pseudo Round-Robin (Pseudo-RR) [138], used for example in Motorola Coldfire 5307.

6.7.1 Related Work

For the purpose of WCET estimation, an exact analysis would require caches with absolutely predictable behaviors. Since several properties of caches, such as associativity and replacement policies, influence predictability, the execution time of a program vary depending both on input data and on the cache hardware state. Therefore, an approximate but safe cache analysis is necessary [108]. The most pessimistic approach, but definitely safe (see Fig. 4.1), assumes that all memory requests result in cache misses, i.e. that a constant timing penalty is always added to the execution time. An alternative method to obtain more precise results is abstract interpretation [50]. The analysis of cache behavior is performed by two different

kind of analysis: the *must analysis* computes a set of memory blocks that are guaranteed to reside in the cache and the *may analysis* computes a set of memory blocks that may be in the cache.

In cache static analysis, no replacement policy can perform better than LRU because it replaces the elements that *has* not been used for the longest time, at the position determined by the associativity k of the cache. For LRU, the *must* analysis is sufficient to obtain precise results about cache hits regardless of program input. Other replacement policies, such as FIFO[56], MRU [106] or PLRU [55], require the combination of *must* and *may* analysis in order to improve the number of predicted hits. In [108] is presented a comparison between replacement policies in terms of well-defined metrics.

The key difference between data cache analysis and instruction cache analysis is the address set. In instruction caches, the address set is singleton because it corresponds to a unique address given by the actual "program counter". Conversely, the address set of a data cache may not be a singleton set, e.g. in the presence array references. Therefore, data cache analysis depends more on the of the spatial locally of a particular program because the data address may change when the same load/store instruction is executed repeatedly. For this reason, some approaches to cache analysis by abstract interpretation consider only instruction caches [50, 137] and some other extend the analysis to unified instruction/data caches [48].

Our approach to *cache analysis* is pragmatic in the sense that our objective is not to propose new abstract domains for cache analysis, but rather to demonstrate that the calculational approach proposed by Cousot in [28] can be used to induce abstract interpreters for cache analysis that match perfectly with the cache analyses found in the abstract interpretation literature. Therefore, and although the ARM9 microprocessor uses RR or FIFO replacement policies, we assume for sake of simplicity that only the instruction memory is cached and that its replacement policy is LRU. Technically, there are also relevant simplifications, since only the *must* analysis is implemented.

6.7.2 LRU Concrete Semantics

We consider a *fully-associative*, or alternatively a *n-way set associative*, instruction cache with the Least Recently Used (LRU) replacement policy [50, 144]. Concrete values in the instruction cache are denoted by a one-to-one correspondence $C \triangleq L \mapsto M$, where C is a fully associative cache with a set of n cache lines $L = \{l_1, \ldots, l_n\}$, each one storing a memory block $m \in M$. The lookup of a cache memory block m inside the cache $c \in C$ is such that $c(l_i) = m$, if m is stored at some cache line $l_i \in L$, or more abbreviately, $c_i = m$.

The lifted domain $M_\perp = M \cup \{\perp\}$ include undefined memory blocks, such that the number of elements inside the undefined cache state $C_\perp$ is always n. Every cached memory block

contains an address *tag* and may be stored in anyone of the n cache lines. Memory blocks are ordered from most- to least-recently-used from left to right. For example, the cache state $[b, e, d, f]$ inside a LRU cache with $n = 4$ cache lines, indicates that b is the most-recently-used element and f the least-recently-used one.

The update of the entire cache set is given by the function $F_C \in M \rightarrow C_\perp \rightarrow C_\perp$.

$$F_C[\![m]\!][c_1, \ldots, c_n] = \begin{cases} [c_i, c_1, \ldots, c_{i-1}, c_{i+1}, \ldots c_n] & \text{if } m = c_i \\ [m, c_1, \ldots, c_{n-1}] & \text{if } \forall i : m \neq c_i \end{cases} \tag{6.24}$$

The correspondent effect of F_C in the denotational setting is specified by the Haskell function f_C. Since the instruction memory is a read-only memory, the specification of the LRU update function does not include the notion of an "evicted" memory block, because it is not necessary to write the replaced cached memory block, c_n, back into the main instruction memory.

```
fC :: Eq a ⇒ a → [a] → [a]
fC m cache = case elemIndex m cache of
                Just i →    let (as, b : bs) = splitAt i cache
                            in b : as ++ bs
                Nothing → let (e : cs) = reverse cache
                            in m : reverse cs
```

The LRU replacement policy can be explained through the notion of *age* of a memory block. The age of a memory m is obtained by counting the number of different memory blocks accessed after the last access to m. In this way, the elements of a LRU cache are ordered by increasing age, from age 1 (most-recently-used) to $n-1$ (least-recently-used). In the case of instruction memories, the oldest element is removed from the cache upon a cache miss.

6.7.3 LRU Abstract Domain

There are only a finite number of cache lines and for each program a finite number of memory blocks. This means the domain of abstract cache states $C^\sharp \triangleq L \mapsto 2^M$ is finite, whence every ascending chain is finite [137]. However, this abstract semantics is distinct from the "collecting semantics" by the fact that the abstract join operator ($\sqcup_C^\sharp$) is different from set union [48]. Instead, the abstract state keeps the notion of relative *age* of a memory block, which is used to identify the cache line in which it is stored. In this way, separate *must* and *may* analysis can be performed by providing proper least upper bound operators. Each of these operators allows the analyzer to lose information that is not relevant for either *may* or *must* analysis, so that the abstract cache states $C^\sharp$ can be reduced to subsets of the collecting cache states.

As described in Fig. 6.1, our approach to WCET analysis perfomers four static analysis simulatenously: *value*, *cache*, *pipeline* and *program flow*. Since the results of the cache analysis are used as execution facts by the pipeline analysis, and since the analysis of the pipeline is guided by full loop unrolling (whence history-sensitive), we only require "local"

results from the cache analysis, results that can be used directly in each fixpoint iteration of the pipeline analysis. Moreover, in particular for LRU, the cache *must* analysis is sufficient to obtain precise results about cache hits. The definition of the *must* least upper bound operator ($\sqcup^\sharp_C$) is:

$$c^\sharp_1 \sqcup^\sharp_C c^\sharp_2 = c^\sharp(l_x) = \{s_i \mid \exists l_a, l_b : s_i \in c^\sharp_1(l_a) \wedge s_i \in c^\sharp_2(l_b), x = max(a,b)\} \tag{6.25}$$

The LRU approach contrasts with many state-of-the-art cache analyses that require both *must* and *may* analysis and compute abstract properties of caches that reach across the entire program, giving rise to categorization of memory accesses such as: *always hit*, *always miss*, *persistent*, *not classified* [142]. However, as will be explained in Section 6.8, a separate cache analysis introduces a certain level of non-determinism in the pipeline analysis because if a memory access is classified as, e.g. *not classified*, both hypothesis of a cache miss or a cache hit must be taken into account. Although our approach is less efficient due to loop unrolling, it produces more precise results because the non-determinism introduced by a separate cache analysis does not exist.

Using the framework of abstract interpretation, an abstract cache state is directly obtained from the concrete domain using a *representation function* $\beta \triangleq C \rightarrow C^\sharp$ [98]. The image of the representation function β are singleton sets of the concrete cache states in its domain. If a memory block $m_x \in M$ is stored in the cache line $l_i \in L$ then β maps a concrete cache state, $c \in C$, to an abstract cache state, $c^\sharp \in C^\sharp$, such that:

$$\beta(c)(l_i) = \{m_x\} \text{ if } c(l_i) = m_x \tag{6.26}$$

As previously explained in the background Section 3.4 on abstract interpretation, when using β it is possible to define a Galois connection and a pair of adjoined functions $\alpha_C : C^\natural \rightarrow C^\sharp$ and $\gamma_C : C^\sharp \rightarrow C^\natural$, such that the abstraction function α_C maps sets of concrete cache states, $C^\natural = \wp(C)$, to their best representation in the domain of abstract cache states, $C^\sharp$. Moreover, the concretization function γ_C is induced by the abstraction function [98] and maps an abstract cache state to the powerset of concrete cache states that is represents.

$$\alpha_C(C^\natural) = \bigsqcup^\sharp_C \{\beta(c) \mid c \in C^\natural\} \tag{6.27}$$

$$\gamma_C(c^\sharp) = \{c \mid \beta(c) \sqsubseteq^\sharp_C c^\sharp\} \tag{6.28}$$

In summary, the abstract domain is formaly defined as a Galois connection by using the least upper bound operator, $\bigsqcup^\sharp_C$, and the partial order, $\sqsubseteq^\sharp_C$, as parameters. The partial order is intuitively induced by the codomain of $C^\sharp$, i.e. by the subset partial order, $\subseteq$, on the powerset domain 2^M, and considering all cache lines in L. Next, we obtain the abstract update function $F^\sharp_C$ using a calculational approach guided by the requirement of monotonocity.

6.7.4 Calculational Design of Abstract Transformer

Similarly to the induction of abstract interpreters for value analysis from the instruction semantics, the abstract interpreter used for cache analysis can be also be obtained by

calculus from the concrete semantics of the LRU cache update cache. Starting from the formal specification $\alpha^{\triangleright}(F^{\natural}_C[\![m]\!])$, where $F^{\natural}_C \in M \to C^{\natural} \to C^{\natural}$ is the forward collection predicate transformer on concrete cache states C, and $\alpha^{\triangleright} \triangleq \alpha_C \circ F^{\natural}_C \circ \gamma_C$, we derive an algorithm $F^{\sharp}_C[\![m]\!]$ satisfying (6.18) by calculus. The bottom element $\bot^{\sharp}_C \in C^{\sharp}$ is a map of size n, where all elements are initialized with the empty set, $\emptyset$. For an abstract state $c^{\sharp} \neq \bot^{\sharp}_C$, we have:

$$
\begin{aligned}
& \alpha^{\triangleright}(F^{\natural}_C[\![m]\!])\, c^{\sharp} \\
= \quad & \wr\, F^{\natural}_C \text{ is the canonical extension of } F_C \text{ to sets} \int \\
& \alpha_C(\{c' \mid \exists c \in \gamma_C(c^{\sharp}) : F_C[\![m]\!](c) = c'\}) \\
= \quad & \wr \text{ def. of the semantic transformer } F_C \int \\
& \alpha_C(\{c' \mid \exists [c_1, \ldots, c_n] \in \gamma_C(c^{\sharp}) : \lambda m.\ F_C(m, [c_1, \ldots, c_n]) = c'\}) \\
= \quad & \wr \text{ def. of the concretization function } \gamma \int \\
& \alpha_C(\{c' \mid \exists [c_1, \ldots, c_n] \in \{c \mid \beta(c) \sqsubseteq^{\sharp}_C c^{\sharp}\} : \lambda m.\ F_C(m, [c_1, \ldots, c_n], m) = c'\}) \\
= \quad & \wr \text{ def. (6.24) of the update function } F_C \text{ for LRU caches} \int \\
& \alpha_C \left(\left\{ \lambda m. \begin{cases} [c_i, c_1, \ldots, c_{i-1}, c_{i+1}, \ldots c_n] & \text{if } m = c_i \\ [m, c_1, \ldots, c_{n-1}] & \text{if } \forall i : m \neq c_i \end{cases} \mid \exists c \in \{c \mid \beta(c) \sqsubseteq c^{\sharp}\} \right\} \right) \\
= \quad & \wr \text{ def. (6.27) of } \alpha_C, \text{ where } \bigsqcup^{\sharp}_C \text{ is the abstract least upper bound operator;} \\
& \quad \text{def. (6.26) of } \beta; \text{ and since } \exists c \in \{c \mid \beta(c) \sqsubseteq c^{\sharp}\} \int \\
& \bigsqcup^{\sharp}_C \left\{ \lambda m. \begin{cases} [c_1 = \beta(m), & \\ \ c_k = c_{k-1} - \beta(m) \mid n = 1 \ldots i-1, & \\ \ c_k = c_k - \beta(m) \mid n = i+1 \ldots n]; & \text{if } m \in c_i \\ [c_1 = \beta(m), c_k = c_{k-1} \text{ for } k = 2 \ldots n] & \forall i : \beta(m) \notin c_i \end{cases} \right\} \\
\sqsubseteq^{\sharp}_C \quad & \wr \text{ Let } c'^{\sharp} \text{ be the least upper bound of the updated states. By the ascending} \\
& \quad \text{chain condition, we have that } c^{\sharp} \sqsubseteq^{\sharp}_C c'^{\sharp} \int \\
& c'^{\sharp} = \lambda m. \begin{cases} [c^{\sharp}_1 = \{m\}, & \\ \ c^{\sharp}_k = c^{\sharp}_{k-1} - \{m\} \mid n = 1 \ldots i-1, & \\ \ c^{\sharp}_k = c^{\sharp}_k - \{m\} \mid n = i+1 \ldots n]; & \text{if } m \in c^{\sharp}_i \\ [c^{\sharp}_1 = \{m\}, c^{\sharp}_k = c^{\sharp}_{k-1} \text{ for } k = 2 \ldots n] & \forall i : \{m\} \notin c^{\sharp}_i \end{cases}
\end{aligned}
$$

Then, the specification of the abstract semantic transformer follows directly:

```
f♯_C :: Eq a ⇒ a → [[a]] → [[a]]
f♯_C m cache = case findIndex (elem m) cache of
                 Just i →    [[m]] ++ foldl remove [] cache
                 Nothing → [[m]] ++ take (length cache − 1) cache
               where
               remove accum cs = accum ++ (delete m cs) : []
```

Although the *cache analysis* is an abstract interpretation that depends exclusively on the abstract domain $C^{\sharp}$, abstract cache states are inquired every time an instruction enters the *Fetch* stage because *pipeline analysis* is performed simultaneously with *cache analysis*.

Therefore, an access function is required to read opcodes from the instruction memory providing, at the same time, a classification of the cache request. To this end, the function *readMemInstrWord* is an interpretation of a memory *Address*, which is stored in the "program counter" register, over an instruction memory of type $\mathbf{I}^\sharp$ that returns the desired **Opcode**, plus a **Classification** that can denote either a cache **Miss** or a cache **Hit**.

```
data Classification = Hit | Miss

readMemInstrWord :: Address → I♯ → (Classification, Opcode, I♯)
readMemInstrWord addr i@I♯ {main = m, cache = c}
 = let opcode = m ! addr
       c' = f♯C (addr, opcode) c
       i' = i {cache = c'}
   in case findIndex (hit addr) c' of
        Nothing → (Miss, opcode, i')
        Just n → let line = c' !! n
                 in (Hit, getOpcode addr line, i')
```

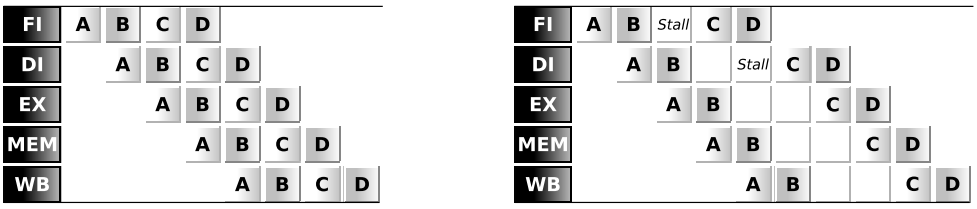
The function *hit* is used to determine if the request memory block is not found in the *cache*. The objective is to go through each *CacheLine* and check if the address *addr* belongs to one of their set of addresses. If the lookup function *findIndex* returns **Nothing**, then the opcode must be read from the *main* store backend. Conversely, if *addr* was indeed found at cache line n, then the opcode is obtained by selecting the proper memory block among the rest of "collected" memory blocks inside that cache line. To this end, the function *getOpcode* was defined. After tracking the *index* inside the list of memory blocks, the opcode is easily obtained by using the function *snd* to select the data block of a *MemoryBlock*.

```
hit :: Address → CacheLine → Bool
hit addr line = elem addr $ map fst line

getOpcode :: Address → CacheLine → Opcode
getOpcode addr line = let index : [] = findIndices (λl → fst l ≡ addr) line
                      in snd $ line !! index
```

6.8 Pipeline Analysis

Pipelining allows overlapped execution of instructions by dividing the execution of instructions into a sequence of pipeline stages PS, and by simultaneous processing N instructions. We consider an ARM9 pipeline with five pipeline stages: *fetch* (*FI*), *decode* (*DI*), *execute* (*EX*), *memory access* (*MEM*), and *write back* (*WB*) [122]. The ideal pipelining is able to fully overlap instructions, as illustrated in Fig. 6.11(a). As opposed to superscalar pipelines, we consider an in-order pipeline, without branch prediction, where only one instruction can be in one particular stage at a given time.

However, total overlapping cannot be reached when pipeline *bubbles* occur, i.e. the situations where pipeline stalls either due to resource conflicts, data dependencies or control flow



(a) Pipeline always filled with instructions

(b) Pipeline with bubbles due to two different kinds of stalling

Figure 6.11: Example of a perfect pipelining on the left (a); and an example of a stalled pipelining on the right (b)

changes. Fig. 6.11(b) illustrates two different reasons for pipeline stalling: the first is caused by the resource conflict resulting from the cache miss when fetching instruction **C** from the instruction memory; the second is caused by a data dependency of the instruction **C** in relation to **A**. In latter case, instruction **C** requires first the instruction **A** to *write back* in order to correctly decode its own operands.

The concrete pipeline semantics present in [122] is a simplified semantics of the processor that describes only the aspects related to its temporal behavior. It considers three types of *hazards* caused by pipeline stalls: (1) the *structural hazards* which are caused by resource conflicts arising when the pipeline does not have enough resources to execute all the possible combinations of instructions without stall; (2) the *data hazards* which are a natural cause of the predetermined order instructions and, consequently, of the logical dependency between the operands of these instructions; and (3) the *control hazards* which arise when the destination address of branch instructions are not resolved early enough to decide which instruction should enter the pipeline next.

In this concrete semantics, details of execution such as register values or results of computations are ignored. On the contrary, our choice to combine value and cache analysis with a pipeline analysis in a single data-flow analysis implies that the semantic transformers defined for the register and memory domains are invoked during pipeline analysis. This also implies that register values and cache contents are updated at the same time as the concrete pipeline state is updated according to the timing model. Nonetheless, the theoretical formalism presented in [122] can easily cope with our through by its definition of *resource association*.

Let $R = \{r_1, \ldots, r_m\}$ be the set of resource types and resources of the processor. Then, a resource association is a pair $(k, \{r_{j_1}, \ldots, r_{j_n}\})$ with $k \in PS$ and $r_{j_1}, \ldots, r_{j_n} \in R$. The set of all resource associations is denoted by $\mathcal{R} = (PS \times 2^R)$. The elements in R can be either static, such as the resource demand of an instruction according to its type, or dynamic when the description of the resource carries its own state. Further, resource associations can be concatenated in sequence along the path of an instruction inside the pipeline, where the stage s and the resource set R evolve accordingly.

Finally, these *resource association sequences* are distinguished by *demand* sequences, which depend on the instruction type, and *allocation* sequences, which carry the current state of the pipeline that is always passed to next stage of a pipeline, including the case where the processing of a new instruction begins.

6.8.1 Semantic Domains

The particularity in our approach to *pipeline analysis* is that the state of the dynamic allocated sequences is simultaneously updated after each pipeline stage, due to the invocation of the abstract state-transformers of the allocated resources in order to obtain the "actual" resource state. These allocated resources are the register and data memory abstract environments as well as the instruction abstract cache contents. For this reason, we re-define the notion of concrete pipeline state in [122] and introduce the notion of an "hybrid pipeline state", which combines *concrete* timing information with the *abstract* state of resources in a single definition.

We define an abstract pipeline state, denoted by $P^\sharp$, as a collection of hybrid pipeline states, P, computed for each program point. This definition corresponds the canonical extension of the hybrid pipeline states to sets of states. As already mentioned in [122], the design of the abstract pipeline domain in this way is enforced by the fact there is not a known abstraction of sets of *concrete* pipeline states. Although the efficiency the *pipeline analysis* highly depends on the number of hybrid pipeline states that must be computed, the termination of the analysis is guaranteed because, for a particular program, there are only finitely many hybrid pipeline states.

Formally, the abstract domain pipeline $P^\sharp$ is defined as the collecting semantics $P^\sharp \triangleq \wp^P$ of hybrid pipeline states. It forms the powerset complete lattice with set inclusion $\subseteq$ as its partial order, set union $\cup$ as its least upper bound, set intersection $\cap$ as its greatest lower bound, $\bot = \emptyset$ as its least element and $\top = \wp^P$ as its greatest element. Consequently, the join of abstract pipeline states is given by set union: $p_1^\sharp \sqcup_P^\sharp p_2^\sharp = p_1^\sharp \cup p_2^\sharp$.

Previously, in Section 6.3, we introduced the notion of store buffers to express the necessity to store intermediate abstract states of the allocated resources allocated during the *pipeline analysis* of every single instruction. Since the least upper bound between the store buffers $R'^\sharp$, $D'^\sharp$, $C'^\sharp$ and $M'^\sharp$ and top level domains $R^\sharp$, $D^\sharp$, $C^\sharp$ and $M^\sharp$ is performed at the level of the abstract pipeline state-transformer, the definition of hybrid pipeline state, P is defined in terms of the store buffers:

$$P \triangleq (\mathit{Time} \times \mathit{Pc} \times \mathit{Demand} \times R'^\sharp \times D'^\sharp \times C'^\sharp \times M'^\sharp \times \mathit{Coord}) \tag{6.29}$$

where *Time* is the global number of CPU cycles, *Pc* is "program counter" of the next instruction to fetch, *Demand* is a 32-bit word used to model the dependences between registers in such a way that each register is either a blocked or unblocked resource, and *Coord* is a

N-sized vector, being N the number of instructions allowed inside the pipeline.

$$Coord \triangleq [TimedTask]_N \tag{6.30}$$

A *TimedTask* is defined for a single instruction, *Instr*, and consists in the current elapsed CPU *Cycles* and the current *Stage* of a given *Task*. A *Task* is associated to one instruction and holds also store buffers inside the "context" of an hybrid state.

$$TimedTask \triangleq (Cycles \times Stage \times Task) \tag{6.31}$$
$$Cycles \triangleq Int \tag{6.32}$$
$$Stage \triangleq FI \mid DI \mid EX \mid MEM \mid WB \tag{6.33}$$
$$Task \triangleq (Instr \times Pc \times Demand \times R'^{\sharp} \times D'^{\sharp} \times C'^{\sharp} \times M'^{\sharp}) \tag{6.34}$$

For the purpose of WCET analysis we are then interested in timing properties of instructions already at the end of the *WriteBack* stage. These properties are measured as CPI (Cycles Per Instruction) and are easily extracted from an hybrid pipeline state by selecting from the *Coord* N-sized vector, the *TimedTask* of the desired instruction (*Instr*) and then extract from it the value of *Cycles* when the stage is *WB*.

The Haskell definitions for the domain definitions are obtained straightforwardly. In order to distinguish the concrete part of an hybrid state we use the parametrized datatype **P** a, where the type variable a denotes a concrete timing *property*. To emphasize that the number N of instructions inside the pipeline is variable, type of the **Coord** coordinates vector is isomorphic to the list type.

```
data P a = P {time :: Int, nextpc :: Word32, demand :: Word32,
              regs :: R♯, datam :: D♯, instrm :: I♯,
              coords :: Coord a}
newtype Coord a = Coord [TimedTask a]
data Stage = FI | DI | EX | MEM | WB
data TimedTask a = TimedTask {property :: a, stage :: Stage, task :: TaskState}
```

As already mentioned, the resource associations is a pair of a stage $s \in PS$ and a set of resources. In our pipeline functional model, a resource association is denoted by **TaskState**, which uses the constructors **Ready**, **Fetched**, **Decoded**, **Stalled**, **Executed** and **Done**, to distinguish the different resource associations inside an allocation resource sequence. Moreover, since we combine the analysis of the resources simultaneously with the *pipeline analysis*, some instances of **TaskState** require also a temporary register file $R^{\sharp}$. The datatype **Reason** is used to specify the cause of a stall. It contains only the constructors for structural and data hazards because the control hazards are handled in a particular way, as will be described latter in this section.

```
data TaskState = Fetched Task R♯ | Decoded Task R♯ | Stalled Reason Task R♯
               | Executed Task R♯
               | Done Task | Ready Task
data Reason = Structural | Data
```

$$\textbf{data Task} = \textbf{Task}\ \{taskInstr :: Instr, taskNextPc :: \textbf{Word32}, taskDemand :: \textbf{Word32},\\ taskRegs :: R^\sharp, taskDmem :: D^\sharp, taskIMem :: \mathbf{I}^\sharp\}$$

The partial order on the domain (**TimedTask** a) is simply the order on natural numbers ($\leqslant$) on its record function *property*. Hence, the partial order on a coordinates vector (**Coord** a) is determined by the maximal element of the N elements of the corresponding list. Finally, the partial order on (**P** a) combines the global timing *time* with the relative elapsed CPU cycles contained in the coordinates vector **Coord**. The partial order on (**P** a) is solely determined by its *concrete* components, defined by a proper instance of the type class *Ord*. The combination of these two timing properties is compared using the componentwise ordering $(p_1, p_2) \sqsubseteq_P^2 (q_1, q_2) \triangleq p_1 \sqsubseteq_P q_1 \wedge p_2 \sqsubseteq_P q_2$.

$$\textbf{instance}\ (Eq\ a, Ord\ a, Num\ a) \Rightarrow Ord\ (\mathbf{P}\ a)\ \textbf{where}\\ compare\ a\ b = compare\ (time\ a, maxCycles\ (coords\ a))\\ (time\ b, maxCycles\ (coords\ b))$$

The maximal element inside a coordinate vector is given by the function *maxCycles*. This function selects the timing *property* of each *TimedTask* using the function *map* and then computes the maximal value using the function *maximum*.

$$maxCycles :: (Ord\ a, Num\ a) \Rightarrow \textbf{Coord}\ a \rightarrow a\\ maxCycles\ (\textbf{Coord}\ vec) = maximum\ \$\ map\ property\ vec$$

6.8.2 Semantic Transformers

We now identify the semantic transformers that perform the *pipeline analysis*. The analysis is performed in three levels: at the lower level, we define the transformer F_T as a morphism on the composite domain *TimedTask*; at middle level, we define the transformer F_P as a morphism on the composite domain P, which uses F_T to compute the new elements inside the N-sized vector *Coord*; finally, at the higher level, we define the transformer $F_P^\sharp$ as a morphism on sets of hybrid states P which uses F_P to create new sets of hybrid pipeline states. The semantic transformers F_P and $F_P^\sharp$ are concisely defined as:

$$F_P \in Instr \mapsto P \mapsto P \tag{6.35}$$

$$F_P(i)(p) \triangleq toContext(i) \circ [F_T \circ fromContext(p)]_N \tag{6.36}$$

$$F_P^\sharp \in Instr \mapsto P^\sharp \mapsto P^\sharp \tag{6.37}$$

$$F_P^\sharp(i)(p^\sharp) \triangleq \{F_P^{5+}(i)(p) \mid p \in p^\sharp\} \tag{6.38}$$

where F_P^{5+} corresponds to the recursive functional application of F_P at least five times, i.e. the number of pipeline stages $k \in PS$. The functions $f_1, f_2, \ldots, f_k, \cdots \in F_T$, specify the effect of pipeline state transformations across a variable number of pipeline steps, which is greater than five in the presence of pipeline *bubbles*. For example, the instruction B in Fig. 6.12 requires l pipeline steps to complete, where $l > k$. The same figure illustrates how a pipeline consists in a sequence of instruction vectors, where each vector is adjoined with a

timing property, $1, 2, \ldots, s, s+1$. This property expresses the relation between the elapsed *cycles per instruction* (CPI) and the current stage of an instruction inside the pipeline.

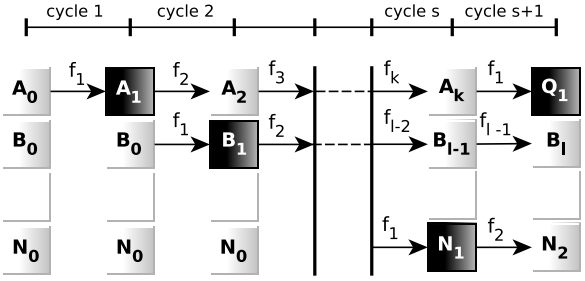


Figure 6.12: Functional overview over pipeline steps

F_P^{5+} does not correspond to the transitive closure of F_P by the fact that *local* worst-case timing properties are always associated to the *WriteBack* (*WB*) pipeline stage of a given task. The reason is that the value and cache analyses are performed simultaneously with the pipeline analysis, thus making the timing analysis a deterministic process for each given input timing property. In this way, intermediate the hybrid pipeline states can be discarded after completion. Let $\{s_k^i \mid k \in PS, k \geqslant 5\}$ be the set of ordered pipeline stages (including stalled stages) required to complete the instruction i. Then, F_P^{5+} is defined by:

$$F_P^{s_{k+1}^i}(i)(p) \triangleq F_P(i)(F_P^{s_k^i}(i)(p)) \tag{6.39}$$

$$F_P^{5+} \triangleq F_P^{s_{WB}^i} \tag{6.40}$$

Example 10. Examples of sets of pipeline state in presence of hazards.
An example of structural hazards is when the next fetched instruction is not found inside the instruction cache. In this case, the timing model adds a timing penalty that corresponds to the read operation on the main instruction memory and subsequent caching of the requested memory address. One example of an extra cache miss penalty is given in Fig. 6.13(a) for the instruction `mov r3, #3` (position 2 in the pipeline). Since the instruction was not found in the cache, the pipeline will advance to the *Decode* stage only after 2 CPU cycles.

Data hazards happen whenever the private data of an instruction is currently being processed inside the pipeline by other instruction. In these cases, the blocked instruction can proceed only after the instruction holding the data has completed the *WriteBack* phase. One example of data stalling is given in Fig. 6.13(b) for the instruction `str r3, [r13]` (position 3 in the pipeline). Since the previous instruction, `mov r3, #3`, is still in the *WriteBack* phase, the decoding of `str r3, [r13]` is stalled. In the next CPU cycle, the data of the register `r3` is signalled as available by the pipeline internal state. Thus, the decoding of `str r3, [r13]` is possible and the pipelining proceeds to the *Execution* phase.

Control hazards happen in circumstances like the one described in Fig. 6.13(c). We known for a fact that the instruction `bgt 8` may change control flow if the branch condition is verified. Thus, the pipeline is flushed so that the next program counter to fetch is available, right after the instruction `bgt 8` writes back.

cycles	N	Next	State
2	0	*DI*	*Fetched*: `mov r13, #0`
0	1	*FI*	*Ready*: `nop`
0	2	*FI*	*Ready*: `nop`

cycles	N	Next	State
3	0	*EX*	*Decoded*: `mov r13, #0`
1	1	*FI*	*Stalled*: `mov r3, #3`
0	2	*FI*	*Ready*: `nop`

cycles	N	Next	State
4	0	*MEM*	*Executed*: `mov r13, #0`
2	1	*DI*	*Feched*: `mov r3, #3`
0	2	*FI*	*Ready*: `nop`

(a) Structural hazard

cycles	N	Next	State
2	0	*DI*	*Fetched*: `ldr r1, [r13]`
6	1	*WB*	*Done*: `mov r3, #3`
3	2	*DI*	*Stalled*: `str r3, [r13]`

cycles	N	Next	State
3	0	*DI*	*Stalled*: `ldr r1, [r13]`
1	1	*FI*	*Stalled*: `mov r2, #0`
4	2	*EX*	*Decoded*: `str r3, [r13]`

(b) Data hazard

cycles	N	Next	State
5	0	*EX*	*Decoded*: `bgt 8`
2	1	*DI*	*Stalled*: `b 20`
1	2	*FI*	*Stalled*: `ldr r1, [r13]`

cycles	N	Next	State
6	0	*MEM*	*Executed*: `bgt 8`
0	1	*FI*	*Ready*: `nop`
0	2	*FI*	*Ready*: `nop`

(c) Control hazard

Figure 6.13: Examples of pipeline states sets in presence of hazards

▲

We now look in more detail at the semantic transformers F_T. The purpose of F_T is to compute the successive intermediate hybrid states inside the pipeline given a N-sized vector of initial resource associations. Each semantic transformation F_T corresponds to a *step* for each instruction inside the pipeline, but since all the N instruction inside the *Coord* coordinates vector share the common context defined in P, it is necessary to update (read/write) the state of the resources in such context. In particular, the value of *Pc* must be known to fetch the next instruction from memory when one instruction inside the pipeline finishes, and the value of *Demand* must be kept updated depending on the blocked/unblocked state of register ports.

The state-transformer F_T is homonymously defined is Haskell by a function of type F_T, which receives as input argument a timing property of type a and the current task state of type **TaskState** and returns the new state of this task on the next pipeline *stage*, instrumented with a new timing *property*. Accordingly, the return type is (**TimedTask** a).

$$\textbf{type}\ F_T\ a = a \rightarrow \textbf{TaskState} \rightarrow \textbf{TimedTask}\ a$$

In order to obtain the effect of the formal specification $\lambda p.F_T \circ \mathit{fromContext}(p)$, we define the function *regular* with the purpose to instantiate a function of type F_T from the context stored inside an hybrid state (**P** a). The additional arguments are the boolean value *hazards*, which specify possible structural or data hazards at a particular *time*, and the current stage. The definition of *regular* is then obtained by pattern matching on the current **Stage**.

When the current stage is **FI** (*Fetch*), the function *regular* is defined as:

```
regular :: (Num a) ⇒ P a → Bool → Stage → F_T a
regular p@P {time = t, nextpc = pc, instrm = i} hazards FI
  = λcycles state → case state of
      Ready task →
        if ¬ hazards
          then let task' = task {taskNextPc = pc, taskIMem = i}
                 in fetchInstr (start t cycles) task'
          else TimedTask {property = cycles, stage = FI, task = Ready task}
      Stalled Structural task buffer →
        let task' = Fetched task buffer
        in TimedTask {property = fetched cycles, stage = DI, task = task'}
```

The resources of interest in the input context of (**P** a) are the global CPU clock cycles, *time*, the "program counter" of the next instruction to fetch, *nextpc*, and the abstract instruction memory, *instrm*. The output of *regular* is an anonymous function of type F_T. Therefore, the function *regular* is a "constructor" of type F_T.

The input argument *state*, of type **TaskState**, holds the abstract state of the allocated resources to the analysis of one instruction inside the pipeline. If this state is the **Ready** state, then two situations can occur during the instruction's opcode: either it was found in the instruction cache or it must be fetched from the main memory. In the former case, the input *task* is updated with the context values *nextpc* and *instrm* and then passed to the function *fetchInstr*, together with the updated timing property. The new timing property is given by applying the function *start* to the context *time* and the input *cycles*. In the latter case, a pipeline bubble is created by letting the task in the same stage **FI** (see Fig. 6.13(a)). Next we given the definition of *fetchInstr*:

```
fetchInstr :: (Num a) ⇒ a → Task → TimedTask a
fetchInstr cycles t@Task {taskNextPc = pc, taskImem = m}
  = let (classification, opcode, m') = readMemInstrWord m pc
        instr = decode opcode
        pc' = pc + 4
        buffer' = setReg♯ bottom R15 (ConcVal pc')
    in if classification ≡ Hit
         then let task' = t {taskInstr = instr, taskNextPc = pc', taskImem = m'}
                in TimedTask {property = fetched cycles, stage = DI,
                              task = Fetched task' buffer'}
         else let task' = t {taskInstr = instr, taskNextPc = pc', taskImem = m'}
                in TimedTask {property = missed cycles, stage = FI,
                              task = Stalled Structural task' buffer'}
```

The function *fetchInstr* is responsible for accessing the instruction memory at the address specified by *nextpc* to obtain an *opcode* used to *decode* an instruction of type **Instr**. Please take note that when a "fetch" operation is performed on the instruction memory, this corresponds to an abstract state-transformation on the instruction cache. Therefore, if the particular

address is not classified as a **Hit** during this particular fixpoint iteration, then the *concrete* timing model of the pipeline will affect the input value *cycles* with the cache "miss penalty" computed by the function *missed*.

When the current stage is **DI** (*Decode*), the function *regular* is defined as:

$$
\begin{aligned}
&\mathit{regular}\ p@\mathbf{P}\ \{\mathit{demand} = d, \mathit{regs} = r\}\ \mathit{hazards}\ \mathbf{DI}\\
&\quad = \lambda \mathit{cycles}\ \mathit{state} \rightarrow \mathbf{if}\ \neg\ \mathit{hazards}\\
&\qquad \mathbf{then\ case}\ \mathit{state}\ \mathbf{of}\\
&\qquad\quad \mathbf{Fetched}\ t\ \mathit{buffer} \rightarrow \mathbf{let}\ \mathit{task}' = t\ \{\mathit{taskDemand} = d\}\\
&\qquad\qquad\qquad\qquad \mathit{buffer}' = \mathit{updateBuffer}\ \mathit{task}\ r\ \mathit{buffer}\\
&\qquad\qquad\qquad \mathbf{in}\ \mathit{decodeInstr}\ \mathit{cycles}\ \mathit{task}'\ \mathit{buffer}'\\
&\qquad\quad \mathbf{Stalled\ Data}\ t\ \mathit{buffer} \rightarrow \mathbf{let}\ \mathit{task}' = t\ \{\mathit{taskRegs} = r, \mathit{taskDemand} = d\}\\
&\qquad\qquad\qquad\qquad\qquad \mathit{buffer}' = \mathit{updateBuffer}\ t\ r\ \mathit{buffer}\\
&\qquad\qquad\qquad\qquad \mathbf{in}\ \mathit{decodeInstr}\ \mathit{cycles}\ \mathit{task}'\ \mathit{buffer}'\\
&\qquad \mathbf{else}\ \ \mathbf{TimedTask}\ \{\mathit{property} = \mathit{dependency}\ \mathit{cycles},\\
&\qquad\qquad\qquad\qquad \mathit{stage} = \mathbf{DI}, \mathit{task} = \mathbf{Stalled\ Data}\ \mathit{task}'\ \mathit{buffer}\}
\end{aligned}
$$

The resources of interest in the input context of (**P** *a*) are the *demand* state of register ports and the state of the regfile that is shared by all elements inside the coordinates vector. Similarly to the previous case, the presence of *hazards* is given by a boolean value. If no **Data** hazard was detected, the state of the **TaskState** is distinguished between a previously **Fetched** task and a previously **Stalled** task. In the former case, it is necessary to update the task resource demand from the context as well as the local buffer from the shared regfile before invoking the function *decodeInstr*.

In the later case, besides updating the local buffer, the update of the task's regfile is also necessary so that all the possible data hazards can vanish. When a **Data** hazard has been was detected, a pipeline bubble is created by letting the task in the same stage **DI** and by computing the penalty associate to a *data hazard* using the function *dependency*. See Fig. 6.13(b), where the analysis of instruction '`str r3, [r13]`' is stalled until the analysis of instruction '`mov r3, #3`' writes back the value of '`r3`'.

$$
\begin{aligned}
&\mathit{decodeInstr} :: (\mathit{Num}\ a) \Rightarrow a \rightarrow \mathbf{Task} \rightarrow R^{\sharp} \rightarrow \mathbf{TimedTask}\ a\\
&\mathit{decodeInstr}\ \mathit{cycles}\ \mathit{task}@\mathbf{Task}\ \{\mathit{taskInstr} = i, \mathit{taskDemand} = d\}\ \mathit{buffer}\\
&\quad = \mathbf{let}\ \mathit{mask} = \mathit{map}\ (\mathit{blocked}\ d)\ (\mathit{operands}\ i)\\
&\qquad\ \ \mathit{stalled} = \mathit{foldl}\ (\vee)\ \mathbf{False}\ \mathit{mask}\\
&\qquad \mathbf{in\ if}\ \neg\ \mathit{stalled}\\
&\qquad\quad \mathbf{then\ TimedTask}\ \{\mathit{property} = \mathit{decoded}\ \mathit{cycles}, \mathit{stage} = \mathbf{EX},\\
&\qquad\qquad\qquad\qquad \mathit{task} = \mathbf{Decoded}\ \mathit{task}\ \mathit{buffer}\}\\
&\qquad\quad \mathbf{else\ TimedTask}\ \{\ \mathit{property} = \mathit{dependency}\ \mathit{cycles}, \mathit{stage} = \mathbf{DI},\\
&\qquad\qquad\qquad\qquad \mathit{task} = \mathbf{Stalled\ Data}\ \mathit{task}\ \mathit{buffer}\}
\end{aligned}
$$

The function *decodeInstr* is mainly responsible for introducing pipeline stalls whenever data hazards occur. The presence of data hazards is detected by checking if any of the *operands* of task's instruction is *blocked* when taking into consideration the *taskDemand* of resources. If some operands are waiting for synchronization with other instructions inside the pipeline,

then the stage of the output **TimedTask** remains **DI**. Otherwise it moves forwards to the **EX** (*Execute*) stage. In the former case, the size of the pipeline bubble is computed in terms of CPU cycles by the function *dependency*. In the latter case, the timing property is updated using the function *decoded*.

When the current stage is **EX** (*Execute*), the function *regular* is defined as:

$$
\begin{aligned}
&\mathit{regular}\ p@\mathbf{P}\ \{\}\ \mathit{hazards}\ \mathbf{EX}\\
&\quad = \lambda \mathit{cycles}\ (\mathbf{Decoded}\ \mathit{task}\ \mathit{buffer}) \rightarrow\\
&\qquad \mathbf{if}\ \neg\ \mathit{hazards}\\
&\qquad\quad \mathbf{then}\ \mathit{executeInstr}\ \mathit{cycles}\ \mathit{task}\ \mathit{buffer}\\
&\qquad\quad \mathbf{else}\ \mathbf{TimedTask}\ \{\mathit{property} = \mathit{conflict}\ \mathit{cycles},\\
&\qquad\qquad\qquad\qquad \mathit{stage} = \mathbf{EX}, \mathit{task} = \mathbf{Stalled\ Structural}\ \mathit{task}\ \mathit{buffer}\}
\end{aligned}
$$

Only structural hazards are considered in the pattern matching with the stage **EX**. These hazards are caused by conflicts in the access to the ALU (Arithmetic Logic Unit) component of the CPU. Control hazards are not excluded, but the resulting pipeline stalls are handled in a different way. As Fig. 6.13(c) shows, when the instruction belonging to the input task is a branch control instruction, e.g. '`bgt 8`', the other $N-1$ instructions inside the pipeline are simply withdrawn from the pipeline because the program counter of the next instruction to fetch from memory cannot be known until the branch instruction is executed and finished. This is done in a post-processing phase of the whole coordinates vector and corresponds one of the effects of the specification function *toContext* of Def. (6.36). The Haskell definition of this specification will be described further ahead. The function *executeInstr* is basically a wrapper of the abstract instruction semantics obtained by calculus, as the example of Section 6.6.4.1 describes when considering the arithmetic instruction '**Add**'.

$$
\begin{aligned}
&\mathit{executeInstr} :: (\mathit{Num}\ a) \Rightarrow a \rightarrow \mathbf{Task} \rightarrow R^\sharp \rightarrow \mathbf{TimedTask}\ a\\
&\mathit{executeInstr}\ \mathit{cycles}\ t@\mathbf{Task}\ \{\mathit{taskInstr} = \mathbf{Add}\ (\mathbf{Reg}\ \mathit{reg1})\ (\mathbf{Reg}\ \mathit{reg2})\ (\mathbf{Reg}\ \mathit{reg3})\}\ \mathit{buffer}\\
&\quad = \mathbf{let}\ \mathit{i2} = \mathit{getReg}^\sharp\ \mathit{buffer}\ \mathit{reg2}\\
&\qquad\ \ \mathit{i3} = \mathit{getReg}^\sharp\ \mathit{buffer}\ \mathit{reg3}\\
&\qquad\ \ \mathit{buffer}' = \mathit{setReg}^\sharp\ \mathit{reg1}\ (\mathit{i2} + \mathit{i3})\ \mathit{buffer}\\
&\qquad \mathbf{in}\ \mathbf{TimedTask}\ \{\mathit{property} = \mathit{executed}\ \mathit{cycles}, \mathit{stage} = \mathbf{MEM},\\
&\qquad\qquad\qquad\ \ \mathit{task} = \mathbf{Executed}\ t\ \mathit{buffer}'\}
\end{aligned}
$$

When the current stage is **MEM** (*Memory*), the function *regular* is defined by:

$$
\begin{aligned}
&\mathit{regular}\ p@\mathbf{P}\ \{\mathit{regs}\}\ \mathit{hazards}\ \mathbf{MEM}\\
&\quad = \lambda \mathit{cycles}\ (\mathbf{Executed}\ t\ \mathit{buffer}) \rightarrow\\
&\qquad \mathbf{if}\ \neg\ \mathit{hazards}\\
&\qquad\quad \mathbf{then\ let}\ \mathit{task}' = t\ \{\mathit{taskRegs} = \mathit{regs}\}\\
&\qquad\qquad\quad \mathbf{in}\ \mathit{memoryInstr}\ \mathit{cycles}\ \mathit{task}'\ \mathit{buffer}\\
&\qquad\quad \mathbf{else}\ \mathbf{TimedTask}\ \{\mathit{property} = \mathit{conflict}\ \mathit{cycles},\\
&\qquad\qquad\qquad\qquad \mathit{stage} = \mathbf{MEM}, \mathit{task} = \mathbf{Stalled\ Structural}\ t\ \mathit{buffer}\}
\end{aligned}
$$

Similar to the previous stage **EX**, structural hazards can happen during the **MEM** stage, e.g. due to a limited number of memory ports. In the absence of a resource conflict, the current task is updated with shared regfile for the reason that the next stage is the final stage **WB**.

This requires that all resources have their state up to date. The function *memoryInstr* is then responsible for using the previously computed memory address values to load/store the previously computed registers values from/to the data memory. Due to space limitations, the reader is referred to the Haskell prototype for the definition of the function *memoryInstr*.

Having defined the constructors of the semantics transformer F_T for each individual instruction, we extend this notion to a vector of N instructions. Since each resource association processed by F_T is one element of the coordinates vector (**Coord** a) inside an hybrid pipeline state (**P** a), the result of each application of F_T is stored into the output coordinates vector. Hence, first is necessary to construct a N-sized vector of F_T functions to which a single hybrid state (**P** a) can be applied. To this end, the function *next* is defined.

```
type F_TArray a = [F_T a]
next :: (Num a) ⇒ P a → F_TArray a
next p@P {coords = Coord vec} = let fs = map (regular p) (hazards vec)
                                in  zipWith (λf task → f (stage task)) fs vec
```

The function space $F_T Array$ is built by *next* in two phases using partial application. First we take the list of *hazards* present in the coordinates vector *vec* to which the function *regular* is partially applied, together with the hybrid state given as a input ("context") argument. In the second phase, with the objective to obtain a list of functions of type F_T, we combine the list of tasks inside *vec* with the list of partially applied functions *fs* using an anonymous function, which takes each function f and *task* and returns f partially applied to the current *stage* of the task.

In this way, the function *next* is an implementation of the high-level specification $\lambda p.[F_T \circ \mathit{fromContext}(p)]_N$. Next, we show how to obtain a state-transformation on input hybrid state (**P** a), by means of the function *step*.

```
step :: P a → F_TArray a → P a
step p@P {time = t, coords = Coord vec} fs
  = let vec' = zipWith (λf t@TimedTask {property, task} → f property task) fs vec
    in p {time = succ t, coords = Coord vec'}
```

The definition of *step* is defined straightforwardly by updating the coordinates vector and the global CPU clock cycles using the successor function. The high-level specification $\lambda i\, p \cdot \mathit{toContext}(i) \circ [F_T \circ \mathit{fromContext}(p)]_N$ is finally implemented by the function F_P. To this end, the context of an hybrid pipeline state needs to be updated after computing a new hybrid state. This is done for each component of the "context" of an hybrid state using the following functions: *regsToCtx* to update of the regfile resource; *memToCtx* to update of the data and instruction memory; *demToCtx* to update of the static resource demand; and *pcToCtx* to update of the next instruction to fetch.

```
F_P :: (Num a) ⇒ Instr → P a → P a
F_P i p = let p' = step p (next p)
          in regsToCtx ∘ memToCtx ∘ demToCtx ∘ pcToCtx ∘ (flush i) $ p'
```

Control hazards are handled before the update of the resources in the context of the output hybrid state by means of the function *flush*. This function takes the input instruction as argument and checks if it is a "branch instruction", for example, '**Bgt**', '**Bne**', etc. In these cases, as Fig. 6.13(c) illustrates, the other instructions inside the pipeline are flushed out the pipeline and replaced by pipeline bubbles to avoid the computation of unnecessary resource allocation sequences.

Having defined the F_P semantic transformer on hybrid pipeline state, the *pipeline analysis* by abstract interpretation is then defined by the abstract state-transformer $F_P^\sharp$. Consider the example instruction **A** in Fig. 6.12. The effect of $F_P^\sharp$ is first to take the "latest" hybrid pipeline state, according to the partial order $\sqsubseteq_P$, containing the **A** in its coordinates vector. Then, this pipeline state is applied to a composition of transformers F_P until the **WB** stage is reached. During the analysis of one instruction, all intermediate states are collected into an output abstract state $P^\sharp$. After completion, the intermediate state are discarded since they are not necessary. In example of Fig. 6.12, the maximum number of CPU cycles necessary to complete **A** is of s cycles.

The Haskell abstract pipeline domain introduced in Section 6.3 is $P^\sharp$. Since the hybrid pipeline states are now parametrized by a type variable denoting timing properties, the definition of $P^\sharp$ must be re-defined. The bottom element and the join operator are defined by an instance of the type class *Lattice*. Since $P^\sharp$ is a powerset lattice, the *bottom* is trivially defined as the empty list, and the *join* operator os defined using the union of lists. Set inclusion is modelled using the predicates *isPrefixOf*, *isInfixOf* and *isSuffixOf*, which determine if the elements of one list form a subset of the other list.

```
newtype P♯ a = P♯ [P a]

instance (Eq a, Ord (P a)) ⇒ Lattice (P♯ a) where
  bottom = P♯ []
  join (P♯ a) (P♯ b) = P♯ (union a b)

instance (Eq (P♯ a)) where
  P♯ a ≡ P♯ b = if isPrefixOf a b ∨ isInfixOf a b ∨ isSuffixOf a b
                then LT else if a ≡ b
                               then EQ else GT
```

As already mentioned, there are two main differences between our approach and the original approach taken by Schneider in [122]. As opposed to the original approach, which is based on previously computed value and cache analyses, our approach which combines the value analysis with the micro-architectural analysis within a single data flow analysis. The second main difference in that our approach distinguishes all program paths introduced by complete loop unrolling, as opposed to [122] which consider a much more reduced number of *path classes* by means of technique based on virtual loop unrolling [86].

Consequently, the efficiency in terms of analysis time and precision of the two pipeline analysis can be considerably different. Although our approach is less efficient in the analysis

of programs with a high number of loop iterations, it is able to automatically compute a program flow analysis as a side effect of the value analysis and to detect infeasible paths. On the other hand, the approach in [122] is more efficient but requires manual annotation of the maximal loop iterations and can be less precise due to the analysis of pipeline states that would not exist in the infeasible program paths.

Last but not least, the number of potential timing anomalies introduced by the analysis is higher in [122] because of the implicit approximations contained in the previously computed cache analysis. Indeed, our approach reduces the number of non-deterministic pipeline sequences by updating the state of cache resource simultaneously with the state of the pipeline, which makes the pipeline analysis of each instruction a deterministic process for each loop unrolling. However, this is not the case for program paths containing an instruction *dominated* by more than one instruction, according to the partial order ($\preceq$) defined for program instructions.

For example, if a memory address is classified as a *cache miss* using the *must* cache analysis [48], this classification can also *contain* those intermediate abstract cache states that were found in the cache but were eventually evicted before reaching fixpoint. In this cases, *scheduling timing anomalies* [107] caused by abstraction introduce non-determinism in pipeline analysis of [122]. Put simply, when the abstraction used for timing analysis introduces non-determinism at a given pipeline stage of a particular instruction, multiple states will be contained in the output abstract pipeline state of that instruction. Whence, multiple path sequences of a finite length of pipeline states need to be combined at those program points where one instruction is dominated by more than one instruction.

Consider the set of all these possible *local* path sequences. A scheduling timing anomaly occurs if we take the worst-case path from this set, i.e the path with highest timing property, and after giving this property as input to the pipeline analysis of the subsequent instructions, the *global* timing property of the complete program path is actually lower than the worst-case execution time of the concatenation of the local paths sequences. In this cases, it is not sound to *compose* the local worst-case timing properties. The solution to this problem is to pass the complete set of hybrid states in the **WB** stage as argument to $F_P^\sharp$.

With our approach, timing anomalies can only occur at merge points of pseudo-parallel program paths, where sets of hybrid pipeline states are combined with set union. Since the local worst-case timing property found in this set may result in global non-worst-case timing properties, the pipeline analysis needs to follow all pipeline sequences from the merge point onwards. However, the algebraic properties of Haskell functions still allow us to use functional composition on $F_P^\sharp$ over any sequence of instructions, by extending the transformer F_P to lists of pipeline states ($\mathbf{P}\ a$). This is achieved using the function *map*, which applies some function to each element of the input list.

After re-defining the abstract value $\mathbb{C}$ with the Haskell datatype ($\mathbf{CPU}\ a$), in order to

include the parametrized pipeline abstract domain, we now give the definition for $F_P^\sharp$. This transformer takes an instruction and a list of hybrid states as arguments and produces a list of pairs of hybrid states and the **Task** with the abstract states of resources computed after the **WB** stage. The intermediate hybrid pipeline states are computed using the function *squash*, which recursively applies F_P until the **WB** stage is detected using the function *done*. Afterwards, the function *keep* discard intermediate hybrid states because they no longer hold relevant data.

```
F_P^♯ :: Ord a ⇒ Instr → [P a] → [(Task, P a)]
F_P^♯ i = map (λs → squash i s)
        where squash i = let go p = let p' = F_P i p
                                    in case done i (step i p) of
                                         Nothing → go p'
                                         Just task → (task, keep p')
              in go
```

Finally, we define the single *dataFlow* propagation, firstly introduced in Section 5.2. The function *updateHS* is used to update the "global" abstract values of type $R^\sharp$, $D^\sharp$, $\mathbf{I}^\sharp$ into the corresponding "buffers" used by the hybrid pipeline states. Then, after applying the abstract transformer $F_P^\sharp$, the "global" abstract values are updated back with the final values of the store "buffers" using the function *updateCPU*.

```
dataFlow :: Ord a ⇒ Instr → (CPU a) → (CPU a)
dataFlow instr cpu = updateCPU cpu $ F_P^♯ instr $ updateHS instr cpu
```

6.9 Summary

This chapter describes the complete structure of the abstract interpretation component of the WCET analyzer, from the points of view of both the formal specification and the corresponding declarative functional definitions. The main objective is to prove that abstract interpreters can be induced by means of a calculational process, in a "correct by construction" fashion, having as starting point a denotational semantics defined for a concrete domain and a Galois connection between the concrete domain and the abstract domain. The induced denotational abstract semantics is in direct correspondence with Haskell function definitions. In this way, we complement the type safety provided by upper-level of our two-level denotational meta-language with correctness by construction of the semantics transformers provided at the lower-level of the meta-language.

Chapter 7

Semantics-based Program Verification

Software development for embedded systems often requires timing validation of hard/ firm real-time constraints. The worst-case execution time (WCET) is often a key parameter in the evaluation of the timeliness of real-time systems and can be used both on the optimization of software and on efficient dimensioning of hardware. Therefore, the access the WCET is desirable at the early stages of software development, independently from the knowledge of a particular compiler or target platform. Moreover, in particular for embedded real-time systems, adaptive configuration mechanisms are often required, e.g. to update available applications after their deployment. Traditionally this is done using a manual and heavyweight process, specifically dedicated to a particular modification. However, to achieve automatic adaptation of real-time systems, the system design must abandon its traditional monolithic and closed conception and allow itself to reconfigure.

An example scenario would be the upgrade of the control software in an automotive embedded system, where an uploaded "patch" code is dynamically linked in the system, but only after the verification of some safety criteria, to prevent security vulnerabilities or malicious behaviors caused by the integration. Nonetheless, the use of the WCET as safety criteria for embedded real-time systems is somehow limited by the computing resources available on the device. Since WCET analysis depend on hardware mechanisms such as cache memories and pipelines, the cost and complexity of the analysis is expected to increase for modern microprocessors. Therefore, the computational burden resulting from the integration of the WCET analyzer into the *trusted computing base* of a distributed embedded system would be unacceptable.

Current solutions for this problem are, among others, Proof-Carrying Code (PCC)[93], Typed-Assembly Languages (TAL)[91] and Abstraction-Carrying Code (ACC)[10]. The common ground of these approaches is the use of some sort of *certificate* that carry verifiable

safety properties about a program to avoid the re-computation of these properties on the consumer side. Recall Fig. 1.3 illustrating the entities of the code "supplier" and the code "consumer" and the role played by the "certificate" in the presence of an untrusted communication channel. The prime benefit of the certificate-based approach is that the computational cost associated to the computation of the safety properties is shifted to the supplier side, where the certificate is generated.

On the consumer side, the safety of the program update is based on a verification process that checks whether the received certificate, packed along with "untrusted" program, is compliant with the safety policy. To be effective, the certificate checker should be an automatic and stand-alone process and much more efficient than the certificate generator. Besides the certificate checking time, also the size of the certificates will determine if the code updating process can be performed in reasonable time.

However, the use of verifiable WCET estimations as safety properties imposes new challenges to the verification process because of the nature of the techniques used to compute the WCET. In fact, since embedded microprocessors have many specialized features, the WCET cannot be estimated solely on the basis of source-code program flow. Along the lines of [142], state-of-the-art tools for WCET computation evaluate the WCET dependency on the program flow using Integer Linear Programming (ILP), while the hardware dependency of the WCET is evaluated using abstract interpretation. By contrast with the kind of tools which are tailored to compute tight and precise WCETs, the emphasis of our verification process is more on highly efficient mechanisms for WCET checking. In this thesis, we propose an extension of the ACC framework with an efficient mechanism to check the solutions of the linear optimization problem.

Besides program verification on the "consumer" side at machine-code level, the objective of this work is also to define a WCET verification mechanism at source-code level, on the "supplier" side. Such mechanism is commonly termed by *back-annotation* and is able to establish a correspondence between the program invariants computed at machine-code level and the program invariants on the source code entities. For example, the invariant of the abstract value of a source code variable would be automatically obtained from the analysis of the machine code if there is the possibility to associate the name of source variable to a memory address. These associations are typically produced during the compilation process and are included in what is designated by *debug information*. The standard DWARF [134] specifies how this debug information is extracted during compilation and is implemented by the majority of the general purpose compilers, e.g. GCC, although with limitations when used with code highly optimized.

The availability of WCET estimates at higher levels of abstraction is a challenging research area that goes along with the present trend for implementing most of the functionality of embedded systems in software, making them more flexible and easily updateable. However,

this challenge imposes new requirements on the data-flow analysis framework in the sense that, besides code generation and static analysis by abstract interpretation, it should be used to express the process of verification of safety properties as well. For this purpose, the algebraic properties of our meta-semantic formalism are very useful, because its higher-order combinators can be composed in a way such that verification conditions are evaluated by an algebraic interpretation of Hoare logic. In this way, we provide a complete tool chain for embedded development where the WCET estimates can be used for verification of real-time requirements, to assist user-driven program optimizations and for evaluation of hardware.

7.1 Transformation Algebra

This section focus on the definition of the transformation algebra referred to in Contrib. (ii). Transformations are performed on dependency graphs and are based on the relational-algebraic properties of the meta-language introduced in Section 5.2. Two transformations are defined: the first takes advantage of the abstract syntax of the intermediate graph language and of the compositional design of the analyzer to compute the global effect of an arbitrary sequence of machine instructions between the input and output points of a sequential subgraph; the second removes the inter-procedural recursion constructors from an intermediate graph. The objective of the former transformation is to reduce the *size* of the ACC certificates and the objective of the later transformation is to minimize the checking *time* of ACC certificates.

7.1.1 Declarative Approach

The design of the upper level of the meta-language by means of a relational algebra provides a compositional framework to express programs as the composition of elementary semantic building blocks. Each building block is represented by a relation τ regarded as a subgraph, with input and output labels, connected inside a dependency graph. The objective of program transformations is to take advantage of the algebraic properties of the meta-language and reduce the number of the connected subgraphs. Program transformations affect both the number of fixed-point iterations in the generation of certificates and the size of the certificates. In the present section, we describe two transformation rules that help in reducing the size of certificates.

In practice, the goal of these transformations is to reduce the number of program points so that the abstract contexts computed by the static analyzer have a smaller number of entries. However, the transformation must preserve the loop bounds computed for the original program at every program label in order to keep the tightness and soundness properties of the WCET. Therefore, in order to inspect the loop bounds computed for the original

program, it is necessary to have access to the labels on the dependency graphs. Indeed, the definition of the intermediate graph language in Section 5.3 follows from this requirement.

A fundamental aspect in the design of the relational-algebraic framework that enables program transformation is the *sequential composition of relations*. As mentioned in Section 5.2, the sequential composition $(\cdot * \cdot)$ of two relations T and R is defined by $a(T * R)c$ iff there exists b such that aTb and bRc. The usefulness of the point-free notation is that the input value a is associated to the right with the output value c, allowing to redefine the sequential composition simply as $(T * R)$, but with type $a \to c$.

In this way, we can compose syntactic phrases of any sequence of relations into a single relation. In terms of fixed-point computation, the interpretation of the relational composition $(\cdot * \cdot)$ in λ-calculus is the functional composition operator $(\cdot \circ \cdot)$. Thus, the interpretation of a relation with multiple syntactical elements uses multiple functional applications in order to obtain a value with the same inferred run-time type (rt). Note that the same reasoning is used in the *functional approach* to interprocedural analysis [128], where a single procedure can be regarded as a "super operation" from the "call" state directly into the "return" state.

A representation for inductive syntactic phrases was given in Section 5.1 by means of the datatype **Expr**. The transition relations τ are denoted by the parametrized datatype (**Rel** a), where the type variable a denotes program state-vectors $\Sigma[\![P]\!]$ and P is a machine program.

```
data Expr = Expr Instr | Cons Instr Expr
data Rel a = Rel (a, Expr, a)
```

With the purpose to reduce the number of program points, we are particularly interested in the simplification of those dependency graphs (of type (**G** a)) composed by two adjacent relations according to the weak topological order. Therefore, candidates for this transformation are instances of subgraphs with type **Seq**, as defined by the pattern matching in the function *reduce*:

```
reduce :: G a → Flow → G a
reduce (Seq (Leaf r) graph) flow
  = let (r', graph') = headG graph
    in if isNothing r'
         then Seq (Leaf r) (reduce graph')
         else  let Rel (b, ir, a) = r
                   Rel (d, ir', c) = fromJust r'
               in if check (a, b, c, d) flow
                    then let r'' = Rel (d, append ir ir', a)
                         in reduce (Seq (Leaf r'') graph') flow
                    else Seq (Leaf r) (reduce graph flow)
```

The transformation is applied only if it does not cause loss of information concerning loop bounds and if it does not affect the instructions used for interprocedural analysis. For this purpose, the function *check* takes as arguments the loop *flow* invariants, which was previously

computed for the original dependency graph, and the tuple with the four program labels, (a, b, c, d), that surround the two candidate relations, r and r'. The transformation is applies only if *all* the three following conditions are satisfied: (1) the loop bounds for these four labels are equal; (2) using the assertion *call*, the candidate relation r' is not a branch instruction or, using the assertion *hook*, the candidate relation r' has a source label that is not the hook point of a procedure call; (3) the top level relation r is not a branch instruction.

$$
\begin{aligned}
&\mathit{check} && :: (\mathbf{St}\ s, \mathbf{St}\ s, \mathbf{St}\ s, \mathbf{St}\ s) \to \mathit{Flow} \to \mathit{Bool}\\
&\mathit{check}\ (a, b, c, d)\ \mathit{flow} = \mathbf{let}\ \mathit{pl} = \mathit{point} \circ \mathit{label}\\
&\qquad \mathit{filter}\ (k1, k2)\ \_ = (k2 \equiv \mathit{pl}\ a) \vee (k2 \equiv \mathit{pl}\ b) \vee\\
&\qquad\qquad (k2 \equiv \mathit{pl}\ c) \vee (k2 \equiv \mathit{pl}\ d)\\
&\qquad l : \mathit{ls} = \mathit{elems}\ \$\ \mathit{filterWithKey}\ \mathit{filter}\ \mathit{flow}\\
&\quad \mathbf{in}\ \mathit{all}\ (\equiv l)\ \mathit{ls} \wedge \neg\ (\mathit{call}\ a) \wedge \neg\ (\mathit{call}\ d) \wedge \neg\ (\mathit{hook}\ a)
\end{aligned}
$$

The selection for candidate relations is done inductively using the function *headG*, which searches for the first (**Leaf** (**Rel** a)) subgraph and its subsequent subgraph. Finally, the function *append* constructs a syntactic phrase by adding a new **Instr** to the **Expr** held by a relation (**Rel** a).

$$
\begin{aligned}
&\mathit{headG} && :: \mathbf{G}\ a \to (\mathit{Maybe}\ (\mathbf{Rel}\ a), \mathbf{G}\ a)\\
&\mathit{headG}\ (\mathbf{Seq}\ (\mathbf{Leaf}\ a)\ \mathit{graph}) && = (\mathbf{Just}\ a, \mathit{graph})\\
&\mathit{headG}\ (\mathbf{Seq}\ a\ b) && = \mathbf{let}\ (r, g) = \mathit{headG}\ a\ \mathbf{in}\ (r, \mathbf{Seq}\ g\ b)\\
&\mathit{headG}\ (\mathbf{Leaf}\ r) && = (\mathbf{Just}\ r, \mathbf{Empty})\\
&\mathit{headG}\ \mathit{df} && = (\mathbf{Nothing}, \mathit{df})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{append} && :: \mathbf{Expr} \to \mathbf{Expr} \to \mathbf{Expr}\\
&\mathit{append}\ (\mathbf{Expr}\ e)\ (\mathit{Instr}\ i) && = \mathbf{Cons}\ e\ (\mathit{Instr}\ i)\\
&\mathit{append}\ (\mathbf{Cons}\ l\ \mathit{ls})\ (\mathit{Instr}\ i) && = \mathbf{Cons}\ l\ (\mathit{append}\ \mathit{ls}\ (\mathit{Instr}\ i))
\end{aligned}
$$

Now considering the ACC scenario in which the static analyzer also runs on consumer sites, a loop transformation can be applied to the control flow graph so that the fixpoint checking is done within a single state traversal. The ACC program transformation, defined by the function *linearize*, is based on two facts: (1) according to the weak topological order, the meta-program contains the part of the loop body subgraph outside the combinator $(\cdot \oplus \cdot)$, so that the loop is analyzed even when the precondition for entering the loop is not satisfied (see Example 6); (2) the static analyzer looks for fixpoint stabilization only at the "head" of the loop (see Example 9). Therefore, all meta-programs on the consumer side are sequential programs after removing the inter-procedural recursive building blocks:

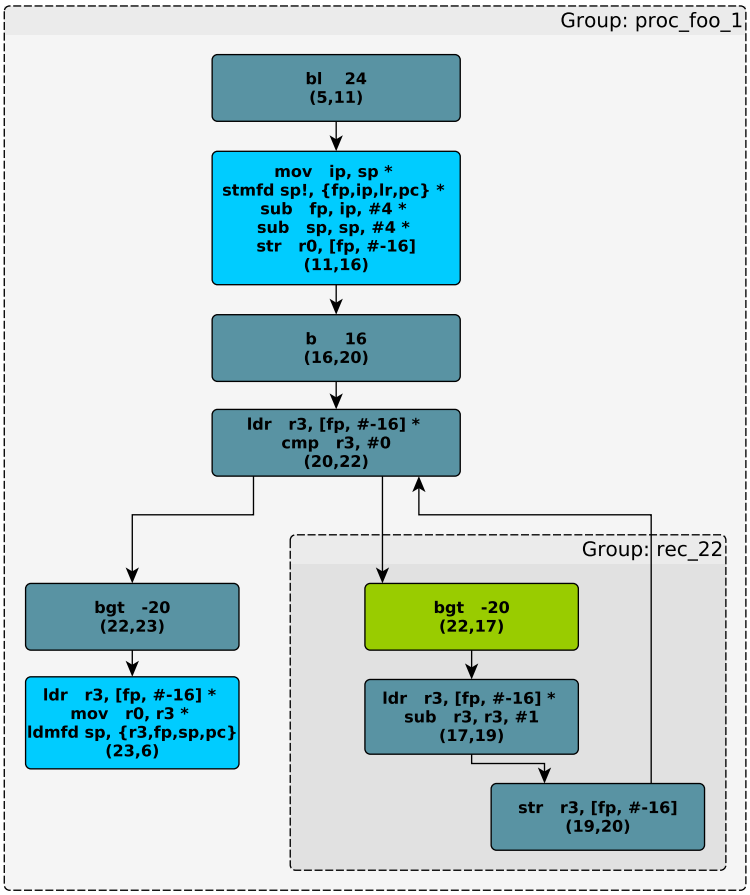
$$
\begin{aligned}
&\mathit{linearize} && :: \mathbf{G}\ a \to \mathbf{G}\ a\\
&\mathit{linearize}\ (\mathbf{Unroll}\ (\mathbf{Leaf}\ r)\ \mathit{graph}) && = \mathbf{Empty}
\end{aligned}
$$

Example 11. Examples of program transformations

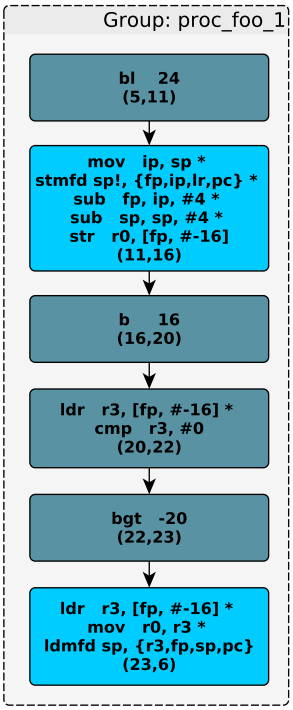
Next, we illustrate the transformations *reduce* and *linearize* applied to the procedure '**foo**' of Fig. 5.2. The dependency graph obtained with *reduce* is given in Fig. 7.1(a). The instructions '`bl 24`' and '`b 16`' are not reduced because they are "branch" instructions. The same applies

for the path-sensitive instruction '`bgt -20`'. Finally, the instruction '`str r3, [fp, #-16]`' is not reduced because there exist multiple nodes with the "sink" label '20' that have different loop iteration counts. All the remaining instructions are sequentially reduced by connecting the corresponding relations.

The dependency graph obtained with the transformation *linearize* is given in Figure 7.1(b). As expected, all nodes in the transformed dependency graph have depth 0 according to the weak topological order. In this way, the fixpoint algorithm is able check the validity of a certificate within a single chaotic iteration. The advantage is that the information contained in the certificate for the nodes left out of the transformed graph consists solely the abstract pipeline states, $P^\sharp$, and *bottom* for the rest of the elements of the abstract domain $\mathbb{C}$.



(a) Transformed DG of the '**foo**' procedure using *transform*



(b) Transformed DG of the '**foo**' procedure using *linearize*

Figure 7.1: Examples of transformations of intermediate graph representations

For the source example in Fig. 5.1, the reduction of the certificate size is shown in Table 7.1. The certificate sent by the code "producer" is processed by the *Sequential Reduction* and by the *ACC Reduction*. After Zip compression, the percentage of size reduction is about 60%. Nevertheless, the size of the certificate (18.3 KBytes) is considerably greater than the size of the source code file (113 Bytes).

Table 7.1: Variation of the certificate size

Original Certificate Fig. 5.7(c)	Sequential Reduction Fig. 7.1(a)	ACC Reduction Fig. 7.1(b)	Compressed Certificate (Zip)
4.5 MBytes	3.6 MBytes	1.2 MBytes	18.3 KBytes

▲

7.2 WCET Verification at Machine Code Level

This section presents Contributions (vi) and (vii). We propose the inclusion of the WCET checking phase inside the ACC framework using the Linear Programming (LP) duality theory. The complexity of the LP problem on the consumer side is reduced from NP-hard to polynomial time, by the fact that LP checking is performed by simple linear algebra computations. Our objective is to demonstrate that the declarativeness of the Haskell programming language is suitable for expressing the LP checking in a very easy, elegant and efficient way (Contribution (x)). Additionally, in one hand, the *flow conservation* constraints of the linear programming problem are formally obtained as abstract interpretations of the relational (natural) semantics of the machine program. On the other hand, the *capacity constraints* of the linear program are taken directly as the result of the *program flow analysis* previously described in Section 6.4.

7.2.1 Related Work

The application of ACC to mobile code safety has been proposed by Albert et al. in [10] as an enabling technology for PCC, a first-order logic framework initially proposed by Necula in [93]. One of the arguments posed by Pichardie et al. [19] in favor of PCC was that despite its nice mathematical theory of program analysis and solid algorithmic techniques, abstract interpretation methods show a gap between the analysis that is proved correct on paper and the analyzer that actually runs on the machine, advocating that with PCC the implementation of the analysis and the soundness proof are encoded into the same logic, giving rise to a *certified* static analysis. Another research project aiming at the certification of resource consumption in Java-enabled mobile devices is Mobility, Ubiquity and Security (MOBIUS) [17]. The logic-based verification paradigm of PCC is complemented with a type-based verification, whose certificates are derived from typing derivations or fixpoint solutions of abstract interpretations. The general applicability of these two techniques depends on the security property of interest.

In ACC [10], verification conditions are generated from the abstract semantics and from a set of assertions in order to attest the compliance of a program with respect to the safety policy. An automatic verifier is able to validate the verification conditions, assuming that the certificate consists of abstract invariant semantics. The consumer implements a defensive

checking mechanism that not only checks the validity of the certificate w.r.t. the program but also re-generates trustworthy verification conditions. Conversely, the abstract safety check in PCC is performed by a first order predicate that checks in the abstract domain if a given safety property is satisfied by the reachable states of the program. If the abstract check succeeds then the program is provably safe, otherwise no answer can be given.

Finally, Albert et al. present in [11] a fixpoint technique to reduce the size of certificates. The idea is to take into account the data flow dependencies in the program and update the fixpoint only at the program points that have their predecessor states updated during the last iteration. In this thesis, the same notion of certificate-size reduction is achieved by means of a program transformation algebra combined with loop unrolling and fixpoint chaotic iterations. The chaotic iteration strategy allows the fixpoint algorithm to look for stabilization at the entry-point of loops for the whole loop to be stable [20]. By the fact that when using meta-language, the first loop iteration is unrolled outside the loop, we apply a program transformation to loop structures that consists in transforming programs with loops in purely sequential programs by keeping only the entry-points of loops.

7.2.2 Declarative Approach

Given a program P, the structure ACC certificates correspond to the map $\Sigma[\![P]\!]$ of Def. (6.2), consisting on the abstract contexts computed during *program flow analysis*, plus the linear programming solutions computed by the primal/dual simplex method on the supplier side. More concretely, given a linear program LP, the solutions of the simplex method are defined by the maps Primal and Dual, which map the variables of LP (Var) to *Double* values ($\mathbb{D}$), plus the solution (WCET) of the objective function.

$$\text{Primal} \in \text{LP} \mapsto \wp(\text{Var} \hookrightarrow \mathbb{D}) \tag{7.1}$$
$$\text{Primal}[\![P]\!] \triangleq \text{Var}[\![P]\!] \mapsto \mathbb{D}$$
$$\text{Dual} \in \text{LP} \mapsto \wp(\text{Var} \hookrightarrow \mathbb{D}) \tag{7.2}$$
$$\text{Dual}[\![P]\!] \triangleq \text{Var}[\![P]\!] \mapsto \mathbb{D}$$

On the consumer side, the verification of abstract contexts, $\text{Invs}[\![P]\!]$, is performed by a single one-pass fixpoint iteration over the machine program P, along the lines of [10], while the LP checking of the Primal, Dual and WCET solutions is based on the duality theory [66]. The idea is that to every linear programming problem is associated another linear programming problem called the *dual*. The relationships between the dual problem and the original problem (called the *primal*) will be useful to determine if the received LP solutions on the consumer side are in fact the optimal ones, that is, the solutions that maximize the objective function (WCET) on the supplier side.

7.2.3 The ILP Verification Problem

The optimization problem is defined as the maximization of the objective function WCET, subject to a set of linear constraints. The variables of the problem are the node iteration variables, x_k, which are defined in terms of the of edge iteration variables, d_{ki}^{IN} and d_{kj}^{OUT}. Edge iteration variables correspond to the incoming (i) and outgoing (j) edges to/from a particular program label identifier k contained in the weak topological order $\langle \text{Lab}, \preceq \rangle$. These linear constraints are called *flow conservation* constraints. Additionally, a set of *capacity constraints* establish the upper bounds, b_{ki} and b_{kj}, for the edge iteration variables.

$$x_k = \sum_{i=1}^{n} d_{ki}^{\text{IN}} = \sum_{j=1}^{m} d_{kj}^{\text{OUT}} \tag{7.3}$$

$$d_{ki}^{\text{IN}} \leqslant b_{ki} \quad \text{and} \quad d_{kj}^{\text{OUT}} \leqslant b_{kj} \tag{7.4}$$

The objective function is a linear function corresponding to the number of node iterations on each label identifier $k \in \mathcal{L}$, weighted by a set of constants, c_k, which specify the execution cost, measured in CPI, associated to every label identifier.

$$\text{WCET} = \sum_{k \in \mathcal{L}} c_k \cdot x_k \tag{7.5}$$

As opposed to similar approaches to WCET analysis, e.g. the combined approach AI+ILP presented in [142], the structure of our optimization problem is particular, in the sense that its solution always assigns integer values to all the variables. This allows us to omit integrality constrains, and furthermore opens the possibility of using linear programming duality in our verification approach. Henceforth, we use refer exclusively linear programming (LP) instead of integer linear program (ILP).

Here, our aim is to demonstrate that the above optimization model can be formally obtained using the theory of abstract interpretation. Note, however, that the *WCET* is not the result of an abstract fixpoint computation. Only the correctness of the LP formulation, which is a set of linear constraints, is expressed by a Galois connection. To this end, the possibility to parametrize the transition system $\langle \Sigma, \tau \rangle$ with different domains is of great importance. Let $\mathcal{T}$ the set of identifiers of program input-output transitions. Then, the flow conservation constraints of Def. (7.3) are a set of equations of type $\wp(\mathcal{L} \mapsto \wp(\mathcal{T}))$. Therefore, a Galois connection $(\alpha_{\mathcal{L}}, \gamma_{\mathcal{L}})$ is established between the relational semantics, $R_{\mathcal{L}}$, and the flow conservation constraints domain, $C_{\mathcal{L}}$:

$$\langle \wp(\mathcal{L} \times \mathcal{T} \times \mathcal{L}), \subseteq \rangle \underset{\alpha_{\mathcal{L}}}{\overset{\gamma_{\mathcal{L}}}{\leftrightarrows}} \langle \wp(\mathcal{L} \mapsto \wp(\mathcal{T})), \leqslant \rangle \tag{7.6}$$

where

$$\begin{aligned} \alpha_{\mathcal{L}}(R_{\mathcal{L}}) \triangleq & \{x_k = \textstyle\sum_{i=1}^{n} d_{ki}^{\text{IN}} \mid \forall x_k \in \mathcal{L} : d_k^{\text{IN}} = \{e' \mid \exists x_l \in \mathcal{L} : \langle x_l, e', x_k \rangle \in R_{\mathcal{L}}\}\} \cup \\ & \{x_k = \textstyle\sum_{j=1}^{m} d_{kj}^{\text{OUT}} \mid \forall x_k \in \mathcal{L} : d_k^{\text{OUT}} = \{e' \mid \exists x_l \in \mathcal{L} : \langle x_k, e', x_l \rangle \in R_{\mathcal{L}}\}\} \end{aligned}$$

$$\gamma_{\mathcal{L}}(C_{\mathcal{L}}) \triangleq \{\langle x_k, d_{\text{out}}, x_l\rangle \mid \quad \exists s_1 \in C_{\mathcal{L}}, \exists d_{\text{out}} \in rhs(s_1) : x_k \in lhs(s_1) \ \wedge \\ \exists s_2 \in C_{\mathcal{L}}, \exists d_{\text{in}} \in rhs(s_2) : x_l \in lhs(s_2) \quad \wedge \ d_{\text{out}} \equiv d_{\text{in}}\}$$

Having proved the correctness of the linear constraints generation process by means of a Galois connection and defining a formal verification of the linear problem based on dual theory, the next step is to encode these mathematical definitions directly into declarative code. The objective of our declarative implementation aims to establish a direct correspondence between formal definitions and functional definitions using the high-level syntax of Haskell.

The desired level of abstraction is achieved by means of a proper *domain specific language*[2] (DSL) that defines a linear (**Program** t) to be a composition of a **direction** of the optimization function, an *objective* function, a set of *variables* and a set of *constraints*. The abstract syntax used to express the objective function and the constrains is defined by the datatype **Expr**. Expressions of the DSL are inductively defined either as constant values (**Con**) of some polymorphic type t, symbolic variables (**Var**), or abstract functions (**App**), which apply the binary operator denoted by *PrimOp* to the first element of a list of expressions ([**Expr** t]) and to the second element of the same list. The abstract functions of the DSL consist in the overloaded infix operators $(+)$, $(-)$ and $(*)$, in the abstract comparisons $(\geqslant^\sharp)$, $(\leqslant^\sharp)$ and $(\equiv^\sharp)$ and, finally, in the abstract assignment $(=^\sharp)$.

```
data Expr t   = Con t | Var Sym | App PrimOp [Expr t]
type PrimOp = String
type Sym    = String
data Direction = Maximize | Minimize
data Program t = Program {direction :: Direction, objective :: (Expr t, String),
                          variables :: [(Sym, String)],
                          constraints :: [(Expr t, String)]}
```

Next, we describe how the formal definitions of the WCET linear program can be defined using the DSL. We start with definition (7.5) of the WCET objective function, which will be maximized by the simplex method. Given a set of labels l, the translation of this definition into the DSL is specified by the function *maximize* as the product of the cost vectors c_k (of type *Flow*) and the the node variable vector x_k, for each k-indexed label. Using the Haskell notation of *list comprehensions*, the DSL objective function is given by:

```
maximize "wcet" $ sum [c (k) * x (k) | k ← l]
```

The next step is the translation of the flow conservation constrains defined in (7.3) and the capacity constraints defined in (7.4). In the former case, we are interested in the translation of the abstraction $\alpha_{\mathcal{L}}(R_{\mathcal{L}})$ of Def. (7.6) into Haskell. To this end, define two sets of constraints using the function *subject_to*: the first for the set of incoming edges and the second for the set of outgoing edges to/from a given node. Besides the set of labels l, also a set of transitions t is available from the monadic context of the DSL.

[2] See the blog *Things that amuse me*: http://augustss.blogspot.fr/search?q=expressions

```
subject_to "incoming" [x (k) =♯ sum [d (j) | j ← t, n (k) ≫♯ e (j)] | k ← l]
subject_to "outgoing" [x (k) =♯ sum [d (j) | j ← t, n (k) ≪♯ e (j)] | k ← l]
```

The auxiliary functions $\ll^\sharp$ and $\gg^\sharp$ respectively determine if the label given by $e\,(j)$ is an `"incoming"` edge or a `"outgoing"` edge of a node. Again, the DSL allows us to define these constraints in a purely declarative way as a direct translation of the mathematical definition. Finally, the capacity constraints defined in (7.4) are translated with another invocation of the function *subject _to*, where a list of j-indexed input-output transition identifiers is taken from the monadic context.

```
subject_to "bounds" [d (j) ⩽♯ b (j) | j ← t]
```

By definition, the variable vector x and the edge vector d must be indexed by unique identifiers. On the other hand, the cost vector c and the bounds vector b are vectors indexed by the number of label identifiers (l) and by the number of transition relations (t). Hence, the variables of the linear problem must be instantiated in the monadic context in such a way that they can be accessed through the same indexes l and t. For this purpose, the function *var_* is used in the **State** monadic context of **Program**:

```
x ← var_ "x" l
d ← var_ "d" t
```

All the required vectors, i.e. the execution costs, the loop iteration bounds and the node and edge names, have the type (*Param_ i*), where i is an index. Since all constraints are specified by means of expressions of type (**Expr** t), we define the parameters of the WCET linear programming specification by instantiating the type variable t with the inductive datatype **LPVal**. The constructor of *Node*, *Cost* and *Bound* constants is **Val**, whereas the constructor *Edge* is **Pair**.

```
data LPVal = Val Int | Pair (Int, Int)
type Param = Expr LPVal
type Param_ i = i → Param
```

The complete DSL *specification* of the WCET linear program in given by:

```
type Cost, Bound, Node, Edge = Param_ String

specification (l :: [String]) (t :: [String]) (c :: Cost) (b :: Bound) (n :: Node) (e :: Edge)
  = do
    x ← var_ "x" l
    d ← var_ "d" t
    maximize "wcet" $ sum [c (k) ∗ x (k) | k ← l]
    subject_to "incoming" [x (k) =♯ sum [d (j) | j ← t, n (k) ≫♯ e (j)] | k ← l]
    subject_to "outgoing" [x (k) =♯ sum [d (j) | j ← t, n (k) ≪♯ e (j)] | k ← l]
    subject_to "bounds" [d (j) ⩽♯ b (j) | j ← t]
```

After specifying the WCET using the linear programming DSL, a further step is required to invoke the GLPK [53] simplex solver. The reason for this is that GLPK is accessible from Haskell through the GLPK bindings library GLPK-hs. However, this requires only a

transformation between the abstract syntax of the DSL into the abstract syntax of GLPK-hs. Afterwards, the system call to the native GLPK simplex solver can be performed. The reader is referred to the Haskell prototype for the complete DSL definition.

7.2.4 Verification Mechanism

Both the objective function and the set of linear constrains can be represented in a matrix form. For this purpose, we need to abstract from the node (x) and edge (d) iterations variables previously defined and consider a single set of variables ($\mathbf{x}$), indexed by non-negative values. In particular, the cost values associated to edge variables are zero in the objective function and the edge iteration bounds ($\mathbf{b}$) are zero for all linear constraints including a node variable.

The equation system of the *primal* problem is defined in terms of the matrix $\mathbf{A}$, with the coefficients of the constraints (7.3) and (7.4), the column vector $\mathbf{x}$ of variables and the column vector $\mathbf{b}$ of capacity constraints. Then, given the row vector $\mathbf{c}$ of cost coefficients, the objective of the primal problem is to maximize the WCET $= \mathbf{cx}$, subject to $\mathbf{Ax} \leqslant \mathbf{b}$. Conversely, the *dual* problem is also defined in terms of the vectors $\mathbf{c}$ and $\mathbf{b}$ plus the matrix $\mathbf{A}$, but the set of dual variables are organized in a complementary row vector $\mathbf{y}$. Then, the objective of the dual problem is to minimize WCET $^{\text{DUAL}} = \mathbf{yb}$, subject to $\mathbf{yA} \geqslant \mathbf{c}$.

Using the simplex method, it is possible compute a feasible solution $\mathbf{x}$ for the primal problem and a paired feasible solution $\mathbf{y}$ for the dual problem. The *strong duality property* of the relationship between this pair of solutions for the purpose of LP checking is: the vector $\mathbf{x}$ is the optimal solution for the primal problem if and only if:

$$\text{WCET} = \mathbf{cx} = \mathbf{yb} = \text{WCET}^{\text{DUAL}}$$

In the ACC setting, this property allows us to use simple linear algebra algorithms to verify the LP solutions that were computed using the simplex method.The verification mechanism is composed by three steps:

1. Use the static analyzer to verify the local execution times included the micro-architectural abstract context. If valid, execution times are organized in the cost row vector **c'**. Then, take the received primal solutions **x'** and solve the equation WCET' = **c'x'** to check if it is equal to the received WCET.
2. Use the static analyzer to verify the loop bounds abstract context. If valid, loop bounds are organized in the row capacities vector **b'**. Then, take the received dual solutions **y'** and verify the strong duality property by testing the equality of the equation **c'x'** = **y'b'**.
3. Extract the coefficients matrix **A'** from the received code and check if the received primal and dual solutions satisfy the equations **A'x'** $\leqslant$ **b'** and **y'A'** $\geqslant$ **c'**. In con-

junction with the two previous steps, this allow us to conclude that **x'** and **y'** are the optimal solutions of the primal and dual problem and, therefore, conclude that the LP verification is successful.

The same approach based on list comprehensions previously used to specify the linear program by means of abstract expressions, can now be used to verify the received solution to the linear program by translating the above three verification steps into equations defined for the *Double* domain *Solution*. The function *checker* performs each of the steps and returns **True** if all steps are successfully evaluated. The variable *same* verifies step 1. The variable *dEqP* verifies step 2. Finally, the conjunction of the variables *axb* and *yac* verifies step 3.

The Primal and Dual maps defined in (7.1) and (7.2), respectively, are encoded as indexable solutions *Solution_*. The received cost vector **c'** and the received capacities vector **b'** are also indexable solutions. The matrix coefficients can be defined either as a list of rows (*rs*) or a list of columns (*cs*), both with the type *RowSolution*, indexed by the row (*r*) or the column (*c*), respectively.

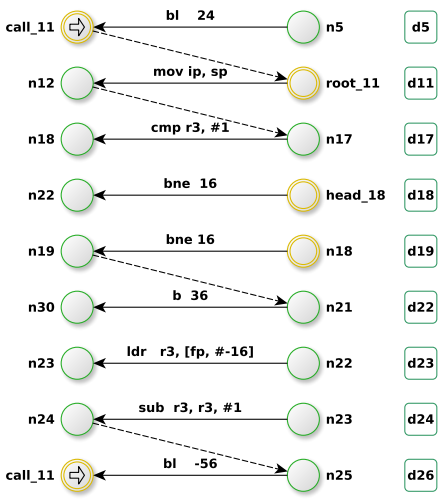
```
type Solution = Double
type Solution_ i = i → Solution
type RowSolution = [(Int, Solution)]
type RowSolution_ i = i → RowSolution

checker (r :: [Int]) (c :: [Int]) (b :: Solution_ Int) (c :: Solution_ Int)
        (rs :: RowSolution_ Int) (cs :: RowSolution_ Int)
        (primal :: Solution_ Int) (dual :: Solution_ Int) (wcet :: Solution)
  = let same = wcet ≡ sum [primal (i) ∗ c (i) | i ← c]
        dEqP = sum [primal (i) ∗ c (i) | i ← c] ≡ sum [dual (i) ∗ b (i) | i ← r]
        axb = and [c (i) ⩽ sum [a′ ∗ dual (j)    | (j, a′) ← k] | i ← c, let k = cs (i)]
        yac = and [b (i) ⩾ sum [a′ ∗ primal (j) | (j, a′) ← k] | i ← r, let k = rs (i)]
    in same ∧ dEqP ∧ (axb ∧ yac)
```

Example 12. Numeric example of a linear programming problem.
Next, we give a numeric example of the LP problem associated to factorial program in Fig. 7.2(a). A subset of the relational semantics of the corresponding machine program is shown in Fig. 7.2(b). For each transition relation, Fig. 7.2(b) includes the name of the edge, indexed to the variable name d, that would correspond to the graph view of the relational semantics. For example, the edge between the nodes "**n5**" and "**call_11**" is called "**d5**".

Table 7.3(a) shows the primal values and execution costs associates to the LP variables (columns in the matrix **A**). For sake of readability, the column **x** displays the node variables (x) plus the un-renamed edge variables (d). As already mentioned, the execution cost associated to edge variables in vector **c** is equal to zero. The column **x*** contains the optimal (primal) solutions for the variable names x_k, where $k \in N$, and for the edge variable names d_{ki}^{IN} and d_{kj}^{OUT}, where $i, j \in E$. Figure 7.3(b) shows the linear equation system from

```
int main(void)
{
  int y = factorial (3) ;
  return y;
}

int factorial (int a)
{
  if (a == 0)
     return 1;
  else
    return (a * factorial (a-1));
}

int foo (int x)
{
  while (x>0)
    x--;
  return x;
}
```

(a) Factorial source code



(b) Relational Semantics

Figure 7.2: Factorial source code and the corresponding relational semantics

Vars (**x**)	Primal (**x***)	Costs in CPU cycles (**c**)
...	–	–
x_{15}	5	7
x_{16}	5	7
x_{17}	5	10
x_{18}	5	8
x_{19}	1	9
x_{20}	1	4
x_{21}	1	6
x_{22}	4	10
x_{23}	4	4
x_{24}	4	10
...	–	–
d_{18}	4	0
...	–	–

(a) Costs and primal values

	Coefficients of variables (matrix **A**)		Constants (**b**)	Dual (**y***)
Flow Conservation	...	=	–	–
	$x_{16} - d_{15}$	=	0	0
	$x_{16} - d_{16}$	=	0	-59
	$x_{17} - d_{16}$	=	0	51
	$x_{17} - d_{17}$	=	0	-51
	$x_{18} - d_{17}$	=	0	42
	$x_{18} - d_{18} - d_{19}$	=	0	-42
Capacities	...	$\leqslant$	–	–
	d_{16}	$\leqslant$	1	-30
	d_{17}	$\leqslant$	2	20
	d_{18}	$\leqslant$	2	-20
	d_{19}	$\leqslant$	1	16
	d_{20}	$\leqslant$	1	-16
	d_{21}	$\leqslant$	1	6
	...	$\leqslant$	–	–

(b) Linear equation system and dual values

Figure 7.3: Numeric example of the LP problem in matrix form

which the coefficients matrix **A** are inferred, and the dual values associated to the rows of **A**. The vector **b** contains the edge iteration upper bounds which are obtained directly from the *program flow* certificate.

To illustrate the definition of a flow conservation constraint, consider the node variable x_{18}. Analyzing Figure 7.2, we know that the input edges to this node is the edge d_{17} and the output edges are simultaneously d_{18} and d_{19}. According to Def. (7.3), these two constraints are shown in the last two rows of the *Flow Conservation* line set in Fig. 7.3(b). The optimal

(dual) solutions are given by the vector $\mathbf{y}^*$. The number of dual solutions is equal to the number of flow conservation constraints plus the number of capacity constraints. ▲

Provided with the primal and dual optimal solutions, the verification mechanism, by means of the function *checker* previously defined, is able to check if the received WCET is in fact the maximal solution of the LP problem, without the need to solve the simplex method all over again.

7.2.5 Verification Time

The verification time of certificates is strongly reduced for the recursive parts of programs, but not for the purely sequential parts of the program. The reason is that chaotic iteration strategy used during fixpoint computation searches for the least fixed point on the supplier side whereas, in the consumer side, the fixpoint algorithm only verifies if the certificate is one post-fixed point [19].

For a purely sequential set of instructions, chaotic iterations are performed using the third equation of Def. (5.18), i.e., in the cases where the previous state value in the certificate is equal to $\bot_{\mathbb{C}}$. In such cases, the transition function is computed exactly once for each of the instructions. On the other hand, during the verification of the certificate, the fixpoint stabilization condition will compare the abstract values $\mathbb{C}$, contained in the received certificate, with the output of the single fixpoint iteration running on the consumer side, in order to check if the certificate is a valid post-fixed point. Consequently, the comparison with $\sqsubseteq_{\mathbb{C}}$ between two abstract values different from $\bot_{\mathbb{C}}$ will take longer to compute than the equality test with $\bot_{\mathbb{C}}$.

For a recursively connected set of instructions, the verification time can be strongly reduced by the fact that the state traversal inside the loop is performed within a single one-pass fixpoint iteration. Two factors contribute for this reduction: (1) the time necessary to compute a valid post-fixed point is much shorter than the time required to perform loop unrolling on the supplier side; (2) with the chaotic iteration strategy, fixpoint iterations over loops are performed only at the "head" of the loop.

Example 13. Comparison between generation/verification times.
Experimental results concerning the checking time of the example in Fig. 7.2(a) are given in Table 7.2 (obtained off-device using an Intel®Core2 Duo Processor at 2.8 GHz). The first parcel is relative to the fixpoint algorithm and the second is relative to the LP simplex method. The checking time of the solutions of the LP problem is close to zero in all cases because the verification mechanism uses simple matrix operations to check that the received solution at consumer sites are indeed optimal ones. As explained before, the performance of the static analyzer is actually worse when the number of instructions outside a loop is

significantly bigger compared to the number of instructions inside loops.

For the source code example in Fig. 7.2(a), when invoking the function '`factorial(4)`', this is specially noticed also due to the sequence of instructions that constitute the epilogue of the recursive function '`factorial`'. However, when invoking the function '`foo`' in the '`main`' procedure, we observe greater reductions of checking time in relation to the generation time for an increasing number of loop iterations.

Table 7.2: Experimental Results

Function Call	Generation Time (sec)	Verification Time (sec)	Ratio (%)
factorial (4)	1.367 + 0.540	1.942 + 0.004	142.0
foo (3)	1.283 + 0.006	1.013 + 0.005	78.9
foo (7)	3.660 + 0.010	2.160 + 0.003	59.0
foo (15)	14.613 + 0.008	4.495 + 0.012	30.7

▲

7.3 WCET Verification at Source Code Level

This section presents Contribution (viii) as the integration of a back-annotation mechanism into the ACC framework with the objective to perform WCET checking on source code level. For this purpose, design contracts are specified in the source code using the expressions of the meta-language parameterized by a simple **while**-like language [101]. These expressions are boolean expressions that compose the effects of state transformations on the information produced at machine-code level on the left-hand side and compare global effect with the provided specification on the right-hand size.

When real-time applications are in high-level programming languages, the loss of information resulting from the compilation to machine code is definitive if is no longer possible to report information about WCET analysis back to the source code level. Existent solutions to this problem either consider that compiler can incorporate the notion of WCET or that the compiler is a black-box component in the WCET toolchain. In both cases, the objective is to provide WCET information at source level by means of an automatic mechanism in order to provide, up to some extent, WCET-driven program verification and optimization.

7.3.1 Related Work

Falk *et al.* present in [46] a compilation process for C programs, designated by WCET-*aware* C *Compiler* (WCC), that incorporates the notion of WCET into the compiler and delegates the WCET analysis on the static analyzer aiT [2]. The advantage of using the aiT tool is the possibility to integrate state-of-art static analysis, namely the analysis of the pipeline behavior and cache structures, into the compiler environment.

Besides being a necessary component for the production of optimized code based on cost

functions, the analysis of the WCET is by itself valuable for the programmer in those cases where the visualization of the WCET at source level is possible. Along these lines, [82] presents a method for *loop unrolling* based on the WCC platform, which optimizes cycles by means of code expansion at the same time that explores maximal reduction of the WCET.

An alternative way to establish the bridge between the analyzed machine code and the high-level representation of source code is to use the DWARF debug information [134] generated by the compiler and included inside the executable binary. Despite the limitations that the DWARF standard reveals when compiler optimizations are active, it allows the integration of a generic compiler into the analysis framework, at the same time that allows WCET data to become visible at the development environment [103].

The main difference between the approach of [46] and the approach of [103] is the granularity of the back-annotation mechanism. While with WCC the data about WCET are exported to the compiler's back-end, which holds an exact correspondence between the source code constructs and the machine code, the use of a generic compiler in the shape of a black-box makes the annotation of WCET data dependent from the DWARF internal representation, which only associates source code lines to the corresponding memory instruction addresses.

Another approach that uses compiler debug information and is based on abstract interpretation is presented in [111]. Here, the main objective is to prove the correctness of the compilation of C source code into ARM machine code by means of a bijection between the invariants computed at source level with the invariants computed at machine level. In particular, the debug information that associates source code variables to memory locations is used to prove the correctness of the machine program in respect to some source code specification.

7.3.2 Back-Annotation Mechanism

The back-annotation mechanism is inspired in the semantics-based program certification framework supported by abstract interpretation presented in [111]. The systematization of the compilation transformation $\mathcal{C}$ and back-annotation $\mathcal{B}$ processes inside this framework opens the possibility to correlate *syntactic* program transformations and *semantic* program transformations within a single framework. Given a source program P, the compiled program M is obtained by applying the syntactic transformation $\mathcal{C}$. Conversely, the syntactic transformation $\mathcal{B}$ exports to source level the abstract semantics computed at machine level.

Semantic transformations are defined either on the level of concrete semantics ($[\![\bullet]\!]$) or abstract semantics ($[\![\bullet]\!]^\sharp$). Fig. 7.4 illustrates the relation between the concrete and abstract semantics for the source program P and the M machine program. The Galois connections $\langle \alpha_P, \gamma_P \rangle$ and $\langle \alpha_M, \gamma_M \rangle$ define the soundness of interpretations in the abstract domain according to the fixpoint approximation theorem [27].

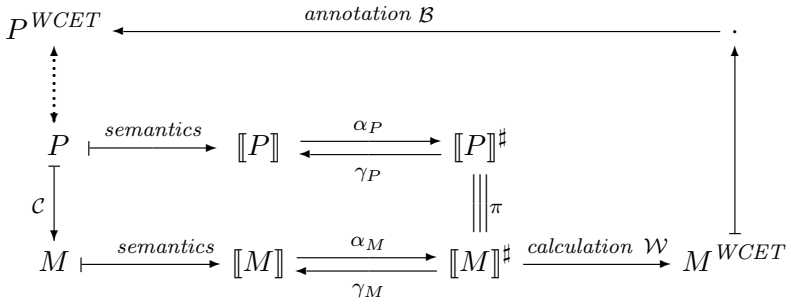


Figure 7.4: Extended abstract interpretation framework with program transformations

Therefore, for the particular abstract domains $P^\sharp$ and $M^\sharp$, the applicability of the methods based on abstract interpretation depends on the existence of the abstraction functions α_P and α_M, respectively. For example, in the context of timing analysis, information about a specific processor hardware is not available at source level and, conversely, program variables are not available at assembly level. Hence, as exposed in Chapter 6, a precise analysis of the WCET can only be performed at hardware level by a combination of a *value analysis*, a *cache analysis* of the contents of the instruction caches and a *pipeline analysis* parameterized by the timing model of the processor.

Nevertheless, using the debug information produced by the compilation function $\mathcal{C}$, it is possible to establish a correspondence between the source and assembly program semantics, defined by relations between source and assembly program points and by relations between program variables and memory locations. Then, the correctness of the compilation function $\mathcal{C}$ is given by the existence of a bijection π between the abstract semantics $[\![P]\!]^\sharp$ and $[\![M]\!]^\sharp$ [111]. Note that this bijection is defined only between the outputs of the *value analysis* of the source code and the machine code, which itself alone is not sufficient for timing analysis. Therefore, we conclude that the abstraction function α_P does not exist for the purpose of timing analysis.

Furthermore, the analysis of the WCET at hardware level requires an additional step to static analysis. If fact, while the techniques of abstract interpretation are capable to determine upper bounds of the local execution times when statically considering all program paths allowed in the program, the identification of the worst-case path depends on the history of computation. Thus, in order to estimate the WCET of a sequence of ordered instructions, a *path analysis* is required. Afterwards, using the compiler debug information and the back-annotation $\mathcal{B}$ function, the fixpoint results of the WCET analysis are associated to the source code program points. Local execution times are extracted from the abstract pipeline domain and become available for each program point and the WCET estimate is available for the main procedure.

As mentioned above, the abstraction function α_P is not defined for the execution timing

cost. Therefore, the diagram in Figure 7.4 does not exhibit the commutation property, due also to the absence of $\mathcal{B}^{-1}$. Let $\mathcal{W}$ denote the function that computes the WCET as the result of a *path analysis*. Then, safe and precise back-annotations P^{WCET} can be formally defined by:

$$P^{WCET} \triangleq \mathcal{B}(\mathcal{W} \circ [\![\mathcal{C}(P)]\!]^\sharp) \tag{7.7}$$

The language of design contracts is based on a simple **while**-like language, extended with some semantic specification constructs, as the BNF specification in Fig. 7.5 shows. The WCET analysis is either executed in *offensive* or *defensive* mode. Support for invariant checking is given in defensive mode by means of the statements $\langle Dbg \rangle$.

$\langle E \rangle \rightarrow int \mid var \mid \langle E \rangle + \langle E \rangle \mid \langle E \rangle$ - $\langle E \rangle \mid$ \var $\mid$ \interval (var) $\mid$ \wcet $\mid [int,int]$
$\langle C \rangle \rightarrow$ **true** $\mid$ **false** $\mid not(\langle C \rangle) \mid \langle E \rangle == \langle E \rangle \mid \langle E \rangle < \langle E \rangle \mid \langle C \rangle \wedge \langle C \rangle \mid \langle C \rangle \vee \langle C \rangle$
$\langle S \rangle \rightarrow var := \langle E \rangle \mid \langle E \rangle \mid$ **return** $\langle E \rangle \mid$ **if** $\langle C \rangle$ **then** $\langle L \rangle$ **else** $\langle L \rangle \mid$
while $\langle C \rangle$ **do** $\langle L \rangle$ **od**
$\langle L \rangle \rightarrow \langle S \rangle \mid \langle S \rangle$**;** $\langle L \rangle$
$\langle Proc \rangle \rightarrow var :=$ **{**$\langle L \rangle$**}** $\mid \langle Dbc \rangle$
$\langle Prog \rangle \rightarrow \langle Proc \rangle \mid \langle Proc \rangle$**;** $\langle Prog \rangle$
$\langle Spec \rangle \rightarrow \langle C \rangle \mid \langle C \rangle$**;** $\langle Spec \rangle$
$\langle Dbc \rangle \rightarrow$ ***** $\langle Spec \rangle$ ***** $var :=$ **{**$\langle L \rangle$**}**

Figure 7.5: Source Language BNF specification

The verification process uses Hoare logic to check a set of assertions in the form of pre- and post-conditions. Instead of using a deductive system, assertions are evaluated by a relational meta-program which encodes Hoare logic is the following way: "if all the pre-conditions evaluate to True then if the program output asserts the set of post-conditions, we conclude that the source code complies with the contract". The procedure together with pre- and post-conditions constitute three subprograms, `program`, `pre` and `post`, which are compositionally combined using our relational algebra in order to obtain the following verification program:

$$\texttt{copy} * (\texttt{pre} \mid\mid (\texttt{program} * \texttt{post})) * \texttt{and}$$

The subprogram `program` encodes the syntactic transformation ($\mathcal{C}(P)$), followed by the semantic evaluation ($\mathcal{W} \circ [\![\mathcal{C}(P)]\!]^\sharp$), as defined in (7.7). The subprograms `pre` and `post` evaluate each assertion in parallel using the pseudo-parallel Haskell combinator (/), and then combine each pair of results with the specialized join operator `and` which encodes the logic function `AND`. The combinator `copy` simply duplicates the input sequentially given to the subprograms `pre` and `program`.

In the source example of Fig. 7.6(a), the contract specification includes a pre-condition, '$\backslash x == 0$', and two post-conditions, '$\backslash interval(x) == [0, 3]$' and '$\backslash wcet <= 300$'. It follows that the design contract is an expression of the meta-language that encodes the Hoare logic of the specification in the following way:

$$\texttt{copy} \;*\; (\backslash x \equiv 0 \;||\; (\texttt{program} \;*\; \texttt{copy} \;*\; (\backslash interval(x) \equiv [0,3] \;||\; \backslash wcet \leqslant 300) \;*\; \texttt{and})) \;*\; \texttt{and}$$

The verification process is essential for an effective use of back-annotations in the development tool chain. Hence, we define post-conditions for the value abstraction, i.e. intervals, and for the WCET cost function. In the former case, the left hand side of the post-condition is encoded by an expression "$\backslash interval(var)$". Similarly, the WCET post-condition is encoded by a boolean expression that compares the constant value "$\backslash wcet$", provided by the back-annotation function $\mathcal{B}$ illustrated in Fig. 7.4, with the desired specification value.

```
1  /*
2  \x==0;
3  \interval(x) == [0,3];
4  \wcet < 300;
5  */
6  int main(void) {
7    int x = 3;
8    while (x>0) {
9      x--;
10   }
11   return 0;
12 }
```

(a) Source Code

Source Line	Program Counter	Label
5	0x33940	0
6	0x33956	4
7	0x33964	6
8	0x33968	7
7	0x33980	10
10	0x33992	13
11	0x33996	14
11	0x33004	14

(b) DWARF Debug Information 1

Variable	Memory Location
x	*stack pointer* + 4 − 20

(c) DWARF Debug Information 2

```
 n1: mov   ip, sp              :root_0    :0x33940
 n2: stmfd sp!, {fp,ip,lr,pc}  :n1        :0x33944
 n3: sub   fp, ip, #4          :n2        :0x33948
 n4: sub   sp, sp, #4          :n3        :0x33952
 n5: mov   r3, #3              :n4        :0x33956
 n6: str   r3, [fp, #-16]      :n5        :0x33960
n10: b     16                 :n6        :0x33964
 n8: ldr   r3, [fp, #-16]      :n7        :0x33968
 n9: sub   r3, r3, #1          :n8        :0x33972
n10: str   r3, [fp, #-16]      :n9        :0x33976
n11: ldr   r3, [fp, #-16]      :n10       :0x33980
n12: cmp   r3, #0              :n11       :0x33984
 n7: bgt   -20                 :head_12   :0x33988
n13: bgt   -20                 :n12       :0x33988
n14: mov   r3, #0              :n13       :0x33992
n15: mov   r0, r3              :n14       :0x33996
exit: ldmfd sp, {r3,fp,sp,pc}  :n15       :0x33400
```

Figure 7.6: (a) source code with a design contract; (b) the GCC DWARF debug information; (c) the relational semantics including "program counter" addresses

Besides the mapping between the program points in the source code and the memory addresses. i.e. "program counter values in Fig. 7.6(b) where the corresponding machine

code is stored, the DWARF debug information also provides the mapping between program variables and the memory locations that hold their values. For the source code in Fig. 7.6(a), this information is displayed in Fig. 7.6(c) and is required to evaluate the post-condition expressed by "*interval(x)*" because it depends on the value of the variable "x".

An example of the use of compiler debug information to determine the correctness of the compilation function $\mathcal{C}$ using abstract interpretation can be found in [111], where is defined a bijection π between the invariants computed at source level ($[\![P]\!]^\sharp$) with the invariants computed at machine level ($[\![M]\!]^\sharp$), as illustrated in Figure 7.4. Despite the fact that the source code is not object of static analysis, the expression "*interval (var)*" used in the design contracts can be used to specify expected behavior of the compilation function, as explained in the next example.

Example 14. Example of a specification of the expected abstract semantics. Consider the case where the abstract semantics $[\![M]\!]^\sharp$ determine that after fixpoint stabilization the value of the *stack pointer* is the decimal 1020. Consulting the table in Figure 7.6(c), we then know that to the variable "x" corresponds the memory location $\mathcal{C}[x] = 1004$. Hence, the abstract interpretation framework presented in [111] provides a semantics-based method to assess the correctness of the compilation $\mathcal{C}$ by checking that the design contract specified at source level is an upper bound of the abstract invariants computed at machine level. In this example:

$$[\![x]\!]^\sharp_{DbC} = [0, 3] \equiv [\![\mathcal{C}[x]]\!]^\sharp_M = [0, 3]$$

▲

However, there may be cases where the abstract semantics computed at machine level is an over-approximation of the concrete semantics of the program. The next example shows the relation between the loss of precision introduced by the static analysis and the expect run-time output of a program that computes the factorial of a number.

Example 15. Example of an over-approximating abstract semantics. Consider the source code in Fig. 7.2(a) containing the recursive definition of the factorial and the corresponding graphical representation of the dependency graph of the machine program in Fig. 7.7.

Two recursive blocks can be found in Fig. 7.7. The first ("rec_18"), corresponds to the instructions inside the recursive call to the '`factorial`' procedure which are executed only if the value of the source variable "a" is different from zero. After the fixpoint stabilization of this subgraph, the fixpoint algorithm proceeds to the second recursive block ("rec_33") which pops the values stored in the memory stack during the first recursive iteration and

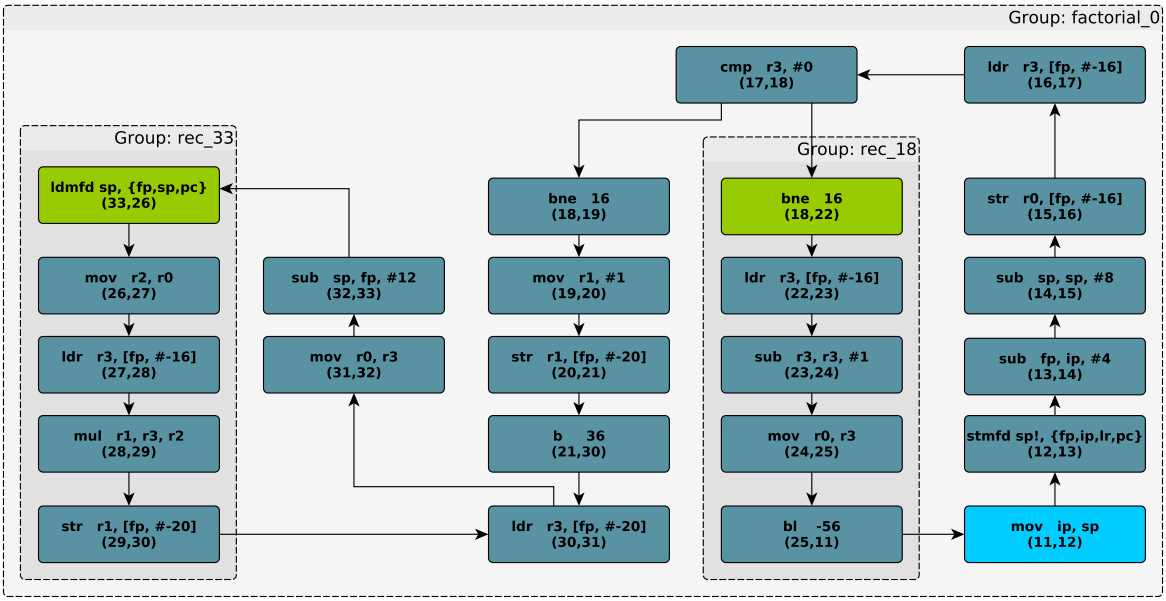


Figure 7.7: Dependency graph of the machine program of the factorial procedure

multiples them, one by one, until the procedure stack is empty.

When the value of the source variable "a" is 4, the number of chaotic fixpoint iterations within the first recursive block is also 4. Examining Fig. 7.7, this means that instructions with states labelled from 11 to 18 are analyzed one more time, which correspond to case where "a" is equal to 0. Consider, as an example, the label 17 and correspond value analysis in Table 7.3. The number of chaotic fixpoint iterations performed at this label is 5 where, after the last iteration, the register `R3` holds the interval [0,4]. For each iteration, it is possible to observe that the "frame pointer" (`R11`) is decreasing at the same time as the recursive calls to '`factorial`' are analyzed. The memory addresses listed in the columns "Value Analysis" hold all the intermediate abstract values computed until fixpoint stabilization for the subgraph "rec_18" is reached.

Table 7.3: Value analysis for label 17 (5 fixpoint iterations)

Label	Value Analysis	Fixpoint Iteration 1	Fixpoint Iteration 2	Fixpoint Iteration 3	Fixpoint Iteration 4	Fixpoint Iteration 5
17	R0	[4,4]	[3,4]	[2,4]	[1,4]	[0,4]
	R1	⊥	⊥	⊥	⊥	⊥
	R2	⊥	⊥	⊥	⊥	⊥
	R3	[4,4]	[3,4]	[2,4]	[1,4]	[0,4]
	R11	0x1000	0x0976	0x0952	0x0928	0x0904
	0x0984	[4,4]	[4,4]	[4,4]	[4,4]	[4,4]
	0x0960	⊥	[3,4]	[3,4]	[3,4]	[3,4]
	0x0936	⊥	⊥	[2,4]	[2,4]	[2,4]
	0x0912	⊥	⊥	⊥	[1,4]	[1,4]
	0x0888	⊥	⊥	⊥	⊥	[0,4]

According to the weak topological order, the fixpoint algorithm, i.e. the reflexive transitive closure of the transition relations in the dependency graph, proceeds towards the second

recursive subgraph "rec_33". Similarly to the first recursive subgraph, the states labelled from 30 to 33 are analyzed one more time when compared to the states contained in this subgraph. For the label 31, Table 7.4 shows the partial analysis results along the required 5 fixpoint iterations until the procedure stack is empty. The invariant associated to the register `R1` is the result of iterating over the memory stack and multiplying the intermediate values computed during the first recursive subgraph. Tables 7.5 and 7.6 show more in detail how the static analyzer proceeds for the labels 28 and 29.

Table 7.4: Value analysis for label 31 (5 fixpoint iterations)

Label	Value Analysis	Fixpoint Iteration 1	Fixpoint Iteration 2	Fixpoint Iteration 3	Fixpoint Iteration 4	Fixpoint Iteration 5
31	`R0`	[0,4]	[0,4]	[0,4]	[0,16]	[0,64]
	`R1`	[1,1]	[1,4]	[1,16]	[1,64]	[1,256]
	`R2`	[1,1]	[1,1]	[1,4]	[1,16]	[1,64]
	`R3`	[1,1]	[1,4]	[1,16]	[1,64]	[1,256]
	`R11`	0x0904	0x0928	0x0952	0x0976	0x1000
	0x0884	[1,1]	[1,1]	[1,1]	[1,1]	[1,1]

Label 28 is associated to the abstract states that exist after the instruction '`ldr r3,[fp #-16]`'. Table 7.5 shows that, according to the value of the "frame pointer" (`R11`), the register `R3` holds the intermediate abstract values first mentioned in Table 7.3. The registers `R0`, `R1` and `R2` carry the invariants resulting from previous factorial multiplication, which are to be used in the next multiplication, as expected for the recursive definition of the factorial.

Table 7.5: Value analysis for label 31 (5 fixpoint iterations)

Label	Value Analysis	Fixpoint Iteration 1	Fixpoint Iteration 2	Fixpoint Iteration 3	Fixpoint Iteration 4
28	`R0`	[1,1]	[1,4]	[1,16]	[1,64]
	`R1`	[1,1]	[1,4]	[1,16]	[1,64]
	`R2`	[1,1]	[1,4]	[1,16]	[1,64]
	`R3`	[1,4]	[2,4]	[3,4]	[4,4]
	`R11`	0x0928	0x0952	0x0976	0x1000

Label 29 is associated to the abstract states that exist after the instruction '`mul r1,r2,r3`'. This instruction multiplies the contents of the registers `R2` and `R3` and stores the result back into register `R1`. In fact, the registers `R0` and `R2` hold the previous iterations results, but the invariant stored in register `R3` is always [1,4]. This means that the final invariant is an over-approximation of the exact value of the factorial of 4. Thus, instead of 24, the analyzer computes an invariant [1,256], which correspond to the exponentiation of 4 by 4.

Although the purpose of the static analyzer is to perform timing analysis, this kind of over-approximation is intrinsic to value analysis by abstract interpretation. Therefore, for the particular of the factorial, the design contract specifying the invariant of the variable "y" in the source code in Fig. 7.2(a) requires the knowledge of the programmer with respect to the machine code implementing the source code. In this case, the specification of the factorial is over-approximated by the specification of the exponentiation.

Table 7.6: Value analysis for label 29 (4 fixpoint iterations)

Label	Value Analysis	Fixpoint Iteration 1	Fixpoint Iteration 2	Fixpoint Iteration 3	Fixpoint Iteration 4
29	R0	[1,1]	[1,4]	[1,16]	[1,64]
	R1	[1,16]	[1,16]	[1,64]	[1,256]
	R2	[1,1]	[1,4]	[1,16]	[1,64]
	R3	[1,4]	[1,4]	[1,4]	[1,4]
	R11	0x0928	0x0952	0x0976	0x1000

▲

7.4 Summary

The novelty of our approach to ACC consists in using the WCET as the safety parameter. A verification mechanism is designed to check that the ACC certificates are valid within a one-pass fixpoint iteration and then check if the WCET is correct using the duality theory applied to linear programming. Experimental results show that the verification process is only efficient when in presence of highly iterative programs

Besides the reduction of verification times, the concept of ACC also requires methods to reduce the size of certificates. We have presented a transformation algebra applicable to dependency graphs to minimize the number of program points considered during fixpoint computations. The simplicity of the process relies on the algebraic properties of the meta-language and on the compositional design of the chaotic fixpoint algorithm.

To facilitate an effective WCET analysis at source-code level, we presented a back-annotation mechanism based on the implementation of the DWARF standard on the compiler side. The back-annotation of the WCET can only be done for a complete program because it is the result of the optimization of a linear program that computes a “global” solution for the maximum execution time. Nonetheless, local execution times can be made available at source level in each source code line.

Chapter 8

Multi-core architectures

The contribution of the work described in this chapter is the extension of our WCET analyzer to multicore systems. As stated in Contribution (ix), we propose a computationally feasible analysis of the WCET in multicore systems by means of the $\mathcal{LR}$-server model presented in [131]. This model defines an abstraction of the temporal behavior of application running on different processor cores and provides a compositional WCET analysis where the same higher-order combinators used to estimate the WCET on single-cores can be combined to perform a sound and efficient timing analysis for multicore systems.

Additionally, Contribution (ix) provides the formalization and implementation of the $\mathcal{LR}$-server model in the context of data flow analysis using the abstract interpretation framework based on Galois connections. Once again in the course of this thesis, we aim at showing that Haskell can be used as a language where the mathematical notions of the $\mathcal{LR}$-server model can be easily, elegantly and efficiently implemented.

WCET estimation of programs running on embedded systems with multicore chips is nowadays one of the most challenging topics in timing analysis because of the intrinsic computational complexity of analyzing multiple processing units sharing common resources. When compared to single-core architectures, the complexity of the timing analysis in multicore environments depends not only on the processor features, but also on the predictability of the timing behavior of each processor when sharing resources, e.g. instruction and/or data memories [40].

In practice, this means that besides the control flow paths through the program, also the "architectural flows", i.e. the number of ways in which a shared resource can be accessed (also called "interleavings"), must be taken into account. Unless shared resources are shared in a composable manner, the different access interleavings allowed by the scheduling arbiter may produce different intermediate hardware states during analysis and, consequently, affect future timing behavior during analysis.

The complexity of the analysis increases exponentially when analyzing architectural flows. Suppose a program that consists of two concurrent processes, P_1 and P_2. The arising conflicts when requesting access to the shared resource are resolved by "interleaving" the execution sequences of the two processes in such a way that either P_1 or P_2 executes by flipping a coin. Hence, for a program with n processes, each one executing a sequence of m instructions, the theoretical number of possible interleavings is $(n.m)!/(m!)^n$. The numerator $(n.m)!$ gives all possible interleavings and the denominator $(m!)^n$ restricts this number to the number of allowed sequences, i.e interleavings that preserve the sequential order in the original machine programs. For realistic programs, the number of execution sequences are huge and although their analysis is a decidable problem, it is not feasible to compute in general.

Consider as an example a tiled multicore with several ARM9 processor cores, shared memories an IO, as shown in Fig. 8.1. Each processor core has an instruction pipeline and an instruction cache memory. Comparatively to single core architectures, the extra source of potential unpredictability is due to interconnect contention originated, e.g. from the shared access to a SRAM shared instruction main memory. As opposed to the timing analysis of a single processor, where the cache miss penalty is constant, in a multiprocessor system each cache miss has a variable penalty, depending on the arbitration protocols specific to the shared resources.

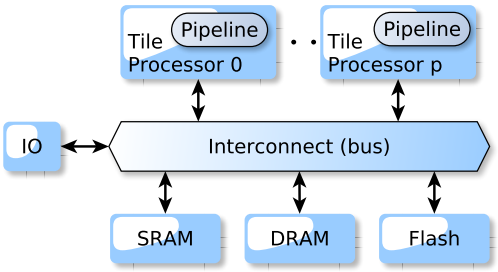


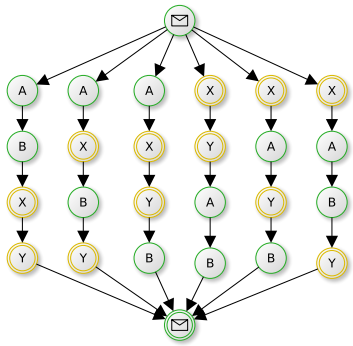
Figure 8.1: Generic multicore architecture

As described in Section 6.8, an ARM9 in-order pipeline allows overlapped execution of instructions by dividing the execution of instructions into a sequence of 5 pipeline stages, and by simultaneously processing a given number of instructions. The static analysis of the pipeline assigns to every program instruction a series of local timing properties, expressing the elapsed *cycles per instruction* (CPI) that are associated to a particular stage of an instruction inside the pipeline. The absence of an abstraction for the *concrete* CPI values in the abstract interpretation literature [122] implies that the *abstract* pipeline domain must be defined as a *set* of pipeline states. For single-core architectures, this does not constitute a computational problem, because there is only a finite, and therefore manageable, number of pipeline states.

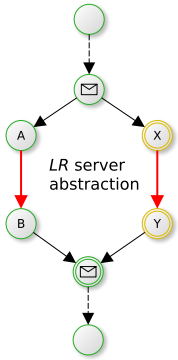
Although the same principles could apply to timing analysis in multicore architectures, the major drawback is having the sets of concrete timing values spread across a huge number of

"architectural flows", a number that is exponentially bigger than the number of control flows and which meaning arises from the following. Without any assumption on the characteristics of the applications running on a multicore system, the execution time of an application running on one processor core may well depend on the activities on the other processor cores. Therefore, the traffic on the interconnect is originated not only from data transfers due to data dependencies between applications, but also from possibly conflicting accesses to shared resources.

Let P_1 and P_2 be two processes running on a homogeneous multicore system comprising two processor tiles. The corresponding number of architectural flows is given in Fig. 8.2(a) and the original control flow is given in Fig. 8.2(b).



(a) Non-compositional timing analysis considering all possible architectural flows between P_1 and P_2



(b) Compositional timing analysis considering only control flows

Figure 8.2: Architectural and control flows for two processes P_1 and P_2, where instructions A and B belong to P_1 and instruction X and Y belong to P_2

Assuming composability in the value domain, i.e. there is no application data shared between processes, the need for timing analysis of architectural flows depends on the scheduling made by the arbiter of the shared resource. Composable arbiters, i.e. arbiters providing complete isolation between application in the temporal domain, analysis of interleavings is not required. An example of such an arbiter is non-work-conserving *time-division multiplexing* (TDM), which statically allocates a constant bandwidth to each processor core. However, when replacing the TDM arbiter by a work-conserving *round-robin* arbiter (RR), the system is no longer composable, since the scheduling of requests depend on the presence or absence of requests from other processor cores. In this case, the analysis of every allowed scheduled sequence in Fig. 8.2(a) must be performed.

However, the analysis of shared resources can be made compositional if the access times are *predictable*. This implies that upper bounds on the access times to shared resources are calculated so that the variation in interference between processor cores visible in Fig. 8.2(a) is removed (abstracted). The formal model of $\mathcal{LR}$ servers [131] is particularly suitable for

determining these upper bounds, since it provides a timing abstraction applicable to most predictable shared resources and arbiters. Fig. 8.2(b) shows how the number of architectural flows is reduced to the number of control flows when abstracting the temporal behavior using the compositional $\mathcal{LR}$-server model.

8.1 Latency-Rate Servers

We now introduce the concept of latency-rate ($\mathcal{LR}$) [131] servers as a shared-resource abstraction. In essence, a $\mathcal{LR}$ server guarantees a processor core a minimum allocated rate (bandwidth), ρ, after a maximum service latency (interference), Θ. As shown in Figure 8.3, the provided service is linear and guarantees bounds on the amount of data that can be transferred during any interval independently of the behavior of other processor cores. The values of the two parameters Θ and ρ depend on the choice of arbiter in the class of $\mathcal{LR}$ servers and its configuration. Examples of well-known arbiters in the class are TDM, *weighted round-robin* (WRR), *deficit round-robin* (DRR) and *credit-controlled static-priority* (CCSP) [6].

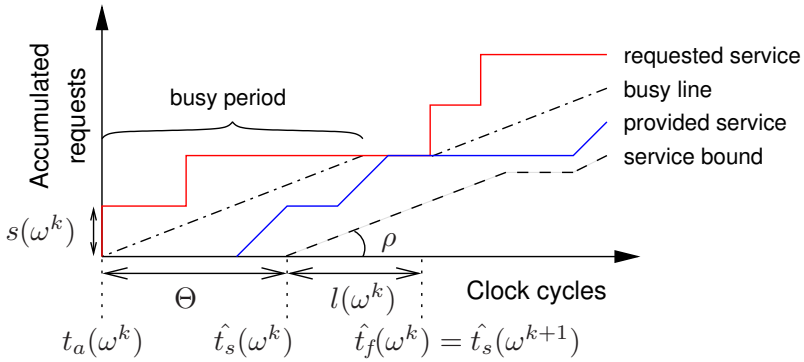


Figure 8.3: A $\mathcal{LR}$ server and its associated concepts.

Like most other service guarantees, the $\mathcal{LR}$ service guarantee is conditional and only applies if the processor core produces enough requests to keep the server busy. This is captured by the concept of *busy periods*, which are intuitively understood as periods in which a processor core requests at least as much service as it has been allocated (ρ) on average. This is illustrated in Fig. 8.3, where the processor core is busy when the requested service curve is above the dash-dotted reference line with slope ρ that we informally refer to as the *busy line*.

We proceed by showing how scheduling times and finishing times of requests are bounded using the $\mathcal{LR}$ server guarantee. From [141], the worst-case scheduling time, $\hat{t_s}$ of the k^{th} request from a processor core, c, is expressed according to Equation (8.1), where $t_a(\omega^k)$ is the arrival time of the request and $\hat{t_f}(\omega^{k-1})$ is the worst-case finishing time of the previous request from processor core c. The worst-case finishing time is then bounded by adding the time it takes to finish a scheduled request of size $s(\omega^k)$ at the allocated rate, ρ, of the processor core, which is called the *completion latency* and is defined as $l(\omega^k) = s(\omega^k)/\rho$.

This is expressed in Def. (8.2) and is visualized for request ω^k in Fig. 8.3.

$$\hat{t_s}(\omega^k) = \max(t_a(\omega^k) + \Theta, \hat{t_f}(\omega^{k-1})) \tag{8.1}$$

$$\hat{t_f}(\omega^k) = \hat{t_s}(\omega^k) + s(\omega^k)/\rho \tag{8.2}$$

8.2 Related Work

Most of existing solutions for timing analysis in multicore systems are restricted to fully timing-compositional architectures, such as ARM7 or ARM9, using predictable arbitration protocols, in particular, the *time-division multiplexing* (TDM) protocol. For example, in [119] an analytical method is proposed to determine upper bounds of the access delays by computing the number of potential conflicts when accessing shared memory and by counting the number of memory accesses possibly generated on different processor cores.

Other approaches perform also schedulability analysis problem in order to find optimal bus scheduling aiming at the minimization of the overall execution time. Examples include the use of ILP in [145], where a series of DSP applications are implemented on several multi-core architectures, based on dynamically reconfigurable processor processor cores, are evaluated in terms of optimal task mapping and scheduling. Another example combines system-level scheduling with timing analysis and a TDMA based bus-access policy, in order to obtain a scheduling and allocation of tasks that guarantees exclusive access and noninterference [116].

An approach to reduce the complexity of potential architectural flows is proposed in [123], exploring the combination of a time-predictable multicore system, using namely a TDM arbiter, with the single-path programming paradigm. In one hand, and by definition, the time-sliced TDM arbitration of the accesses to shared memories provides time-predictable memory load and store instructions. On the other hand, single-path programming avoids control flow dependent timing variations by keeping the execution time of tasks constant, even in the case of shared memory access of several processor cores. The resulting code from the single-path transformation has exactly one execution trace that all executions of the task have to follow because the tasks on the cores are synchronized with the time-sliced memory arbitration unit. This eliminates the need for dynamic conflict resolution and guarantees the temporal isolation, i.e. composability, of multicore programs.

Recent work on abstract interpretation-based timing analysis of TDM shared buses connecting processor cores to a shared main memory has been given in [71] as the combination of loop unrolling [14] and time-alignment of each loop iteration [24]. The first approach tries to determine the precise time at which every single memory access takes place, which requires the analyis to unroll all loops virtually to determine the access times for each access individually. The second approach eliminates the dependency on the number of loop iteration of [14] by aligning each loop head execution to the first TDM slot during the analysis, which

results in an additional penalty to be added in WCET estimation. The solution proposed in [71] is almost as precise as [14] and only slightly less efficient than [24].

The unconstrained use of shared caches in a multicore system makes the cache static analysis a nearly impossible task, even when using caches with LRU replacement strategy [143]. In fact, when concurrent processes running on different processor cores shared data and instruction caches, set of potential interleavings of the running threads result in a huge state space to be explored, which inevitably leads to poor precision. Attempts to solve this problem include a recent approach [78], in which concurrent program are modeled as graphs of *message sequence charts* that capture the ordering of computation tasks across processes. The proposed timing analysis iteratively identifies tasks whose lifetimes are disjoint and uses this information to rule out cache conflicts between certain task pairs in the shared cache. This approach is flow insensitive but mildly context sensitive as it considers instructions in loops and not in loops differently. Therefore, the obtained precision is not considered satisfactory.

The principle *architecture follows application* advocated in [40] points to a scenario where interconnect buses should be "used for the communication between cores and the shared cache and memory in such a way that they have deterministic access times despite being global and shared." The objective is to perform scheduling and allocation of tasks so that exclusive access and noninterference is guaranteed. In this direction, [60] proposes the elimination of the interference altogether by exploring different scenarios of locking and partitioning the shared cache, using a mechanism of cache bypassing to eliminate the cache conflicts between different processor cores.

The objective of our WCET analysis on multicore systems aims at surpassing the intrinsic computational complexity of timing analysis of multiple processing units sharing common resources and extend the range of supported arbitration protocols. For this purpose, we propose a novel application of *latency-rate* ($\mathcal{LR}$) servers [131], phrased in terms of abstract interpretation, to achieve timing compositionality on requests to shared resources. The $\mathcal{LR}$ server abstraction is a simple linear lower bound on the service provided by a resource. The model was originally developed for analysis of networks, but has gained popularity in the context of real-time embedded systems in recent years. Example uses of the model involve modeling buses [139], networks-on-chips [58], and SRAM and SDRAM memories [7]. The main advantage of the $\mathcal{LR}$ abstraction is the ability to perform compositional timing analysis of any arbiter belonging to the class of $\mathcal{LR}$ servers.

8.3 Calculational Approach to Architectural Flows

Our approach to static timing analysis for multicore systems reuses the two-level denotational meta-language [113, 115] described in Section 5.2 by extending the intermediate graph

language described in Section 5.3. In summary, the intermediate graph language is used to represent all the program paths allowed to execute as a mimic of the execution order of program inferred from the program structure known at compile time, as described in Section 5.1. The basic element of the intermediate language is the **Leaf** constructor, which holds an input-output transition relation $\tau \subseteq (\Sigma[\![P]\!] \times \text{Instrs}[\![P]\!] \times \Sigma[\![P]\!])$, of type $(\mathbf{Rel}\ a)$, where the type variable a denotes a program state $\Sigma[\![P]\!]$, of type $(\mathbf{St}\ a)$, where P is a machine program.

In order to represent the notion of architectural flows, we extend the intermediate graph language with the constructor **Conc**, denoting two subgraphs statically assigned to run on two different processor cores.

$$\begin{aligned}\mathbf{data}\ \mathbf{G}\ a = \mathbf{data}\ \mathbf{G}\ a = \mathbf{Empty} \mid \mathbf{Leaf}\ (\mathbf{Rel}\ a) \mid \mathbf{Seq}\ (\mathbf{G}\ a)\ (\mathbf{G}\ a) \mid \mathbf{Unroll}\ (\mathbf{G}\ a)\ (\mathbf{G}\ a)\\ \mid \mathbf{Unfold}\ (\mathbf{G}\ a)\ (\mathbf{G}\ a) \mid \mathbf{Choice}\ (\mathbf{Rel}\ a)\ (\mathbf{G}\ a)\ (\mathbf{G}\ a) \mid \mathbf{Conc}\ (\mathbf{G}\ a)\ (\mathbf{G}\ a)\end{aligned}$$

In the same way, the calculation of architectural flows is performed by taking advantage of the algebraic properties of the higher-order relational combinators of the two-level denotational meta-language in order to generate abstract interpreters in the compositional form of Def. (5.23). Accordingly, Haskell fixpoint interpreters are automatically generated by providing type safe interpretations to the higher-order relational combinators into the λ-calculus.

The main advantage of the higher-order relational combinators defined by the type system of Def. (5.19), is that new functions can be obtained throughout the composition of more basic functions, in analogy to graph-based languages. Therefore, as for any well-typed meta-program, the calculation of architectural flows will result in meta-programs with the unified type $(\mathbf{St}\ a \rightarrow \mathbf{St}\ a)$. Consequently, the analysis of architectural flows is a decidable problem. However, by the reasons already mentioned in Chapter 5, it is not feasibly computable in general. Even so, meta-programs denoting fixpoints of architectural flows can be automatically obtained by means of the function *derive*.

$$\begin{aligned}&\mathit{derive} :: (\mathit{Infeasible}\ (\mathbf{St}\ a), \mathit{Iterable}\ (\mathbf{St}\ a), \mathit{Lattice}\ a, \mathit{Lattice}\ (\mathbf{St}\ a),\\ &\qquad \mathit{Transition}\ (\mathbf{Rel}\ (\mathbf{St}\ a)), \mathit{Synchronizable}\ (\mathbf{St}\ a), \mathit{Eq}\ a) \Rightarrow\\ &\qquad (\mathbf{St}\ a \rightarrow \mathbf{St}\ a) \rightarrow \mathbf{G}\ (\mathbf{St}\ a) \rightarrow (\mathbf{St}\ a \rightarrow \mathbf{St}\ a)\\ &\mathit{derive}\ f\ (\mathbf{Conc}\ a\ b) = \mathbf{let}\ \mathit{is} = \mathit{interleavings}\ a\ b\\ &\qquad\qquad\qquad\qquad\qquad\qquad ms = \mathit{map}\ (\mathit{derive}\ (\mathit{create}\ b))\ \mathit{is}\\ &\qquad\qquad\qquad\qquad\qquad\mathbf{in}\ f * \mathit{scatter}\ (\mathit{length}\ ms) * (\mathit{distribute}\ ms) * \mathit{reduce}\end{aligned}$$

The meaning of a subgraph $(\mathbf{Conc}\ a\ b)$ is given by the composition of the current "continuation", f, with the whole set of *interleavings* between a and b. The creation and synchronization of these two processes is modeled by the *scatter*/*reduce* computational pattern, commonly used in parallel computing. Inductively, the derivation of each individual trace is accomplished by using *derive* with the initial "continuation" returned by the function *create*, which is defined in type class *Synchronizable*.

```
class Synchronizable a where
  create    :: G a → a → a
  continue  :: Rel a → a → a
  break     :: Rel a → a → a
```

The function *interleavings* is used to obtain the set of architectural flows of Fig. 8.2(a). This function takes two dependency subgraphs and returns a list of subgraphs. Using list comprehensions, the allowed sequences are a subset of all *permutations* of the transition relations belonging to both processes, *p1* and *p2*. After converting dependency graphs into list using the function *toList*, the illegal sequences contained in the list of permutations are removed by means of the constraint *preserve*, which excludes any *generated* sequence that, after being filtered from the transition relations belonging to the other process, is not exactly equal to the *original* sequence.

```
interleavings :: G (St a) → G (St a) → [G (St a)]
interleavings p1 p2
  = let preserve original generated = original ≡ filter ((flip elem) original) generated
        (p1L, p2L) = (toList p1, map (colorfy Interleaved) (toList p2))
        sequences = [is | is ← permutations (p1L ++ p2L),
                                  preserve p1L is, preserve p2L is]
        ts = map traces (groups sequences)
    in map (foldl interleave main) ts
```

Additionally, the new constructor **Interleaved** is defined as the syntactical element representing an interleaved instruction. To change the "constructor" of an instruction, the function *colorfy* was defined. Afterwards, the selection of input-output relation holding interleaved instructions is obtained using the function *isInterleaved*. Henceforth, the reader is referred to the Haskell prototype [140] for the omitted function definitions.

```
data Expr = Expr Instr | Interleaved Instr
          | Cons Instr Expr
```

After the computation of the interleaved sequences, it is necessary to transform these sequences back into dependency graphs. To this end, the functions *groups*, *traces* and *interleave* are defined according to the encoded logic in the datatype (**G**(**St** *a*)), so that each architectural flow can be instantiated as set of connected transition relations, which possibly pertain to different applications, running on different processor cores.

The effect of the function *groups* is to identify the program labels where new execution sequences to the *p2* process are interleaved with execution sequences of the *p1* process. To this end, the type class *Synchronizable* defines the function *break* to detect the "interleaved" program labels, using the function *isInterleaved*. In this way, the input dependency graphs, which are given as an ordered list of relations (i.e. traces), are transformed into a list of "tagged" traces.

Afterwards, using of a folding mechanism, the function *traces* traverses lists of "tagged" traces, i.e. sequential interleavings, in order to instantiate dependency graphs of type

(**G**(**St** a)). Finally, the "tagged" dependency graphs labelled with the program label identifiers where executions sequences of *p2* are interpolated inside the *p1* execution sequences, are transformed into "plain" dependency graphs by means of the function *interleave*. This step is repeated for all traces, *ts*, in the last line of the function *interleavings*, then completing the calculation of all traces' dependency graphs.

Back to the definition of *derive* for dependency graphs of type (**Conc** a b), its purpose is to calculate meta-programs describing the whole set of architectural flows the subgraphs a and b. As described in Fig. 8.2(a), the creation and posterior synchronization of the analyses of these two subgraphs is modeled using the general *scatter*/*reduce* computational pattern. Each individual trace is derived passing as the initial "continuation" the function *create* applied to the subgraph b, stating that every time the *p2* process begins, it starts with the hardware state provided by the function *create*. This function initializes the pipeline of the processor dedicated to the analysis of b with its next program counter address.

At this point, the datatype (**CPU** a) must be re-defined to include the notion of a *shared* abstract instruction memory ($\mathbf{I}^\sharp$), a set of *multi*-processor cores and a flag stating the *active* core. The multiple processor cores are denoted by $MultiCore$, which maps core identifiers to values of type (**Core** a). Similarly to the WCET analysis for single-cores, each **Core** comprises an abstract register environment memory ($R^\sharp$), an abstract data memory ($D^\sharp$) and a pipeline abstract domain memory ($P^\sharp$).

$$\begin{aligned}
&\textbf{data CPU } a = \textbf{CPU } \{shared :: \mathbf{I}^\sharp, multi :: MultiCore\ a, active :: Int\} \\
&\textbf{type } MultiCore\ a = Map\ Int\ (\textbf{Core } a) \\
&\textbf{data Core } a = \textbf{Core } \{registers :: R^\sharp, , dataMem :: D^\sharp, pipeline :: P^\sharp\ a\}
\end{aligned}$$

The definition of *create* is the following. The processor core analyzing the dependency graph *p2* is initialized with the identifier '*1*'. Given an input state, the *value* of the hardware state stored in the *invs* abstract context at the program *point* given by the *source* of the first input-output relation of the *p2*, is modified using the function *resetPipelineAt*. The main purpose of this function is empty the pipeline of the processor core with identifier '*1*' and set its next "program counter" to $base * 4$.

$$\begin{aligned}
&create :: \mathbf{G}\ (\mathbf{St}\ (\mathbf{CPU}\ a)) \rightarrow (\mathbf{St}\ (\mathbf{CPU}\ a)) \rightarrow (\mathbf{St}\ (\mathbf{CPU}\ a)) \\
&create\ p2\ s@\mathbf{St}\ \{invs = i\} \\
&\quad = \textbf{let } base = (point \circ source \circ head \circ toList)\ p2 \\
&\qquad\quad node = i\ !\ base \\
&\qquad\quad cpu' = resetPipelineAt\ 1\ (value\ node)\ (base * 4) \\
&\qquad \textbf{in } s\ \{invs = insert\ base\ (node\ \{value = cpu'\})\ i\}
\end{aligned}$$

After the initialization of the processor core running the *p2*, we proceed with the calculation of each individual interleaving that will receive this new hardware state as input value. This is done in the last line of the function *derive* with the meta-program $f * scatter\ (length\ ms) * (distribute\ ms) * reduce$, where ms is a list of interleavings. The reasoning expressed in this meta-program is simply to *scatter* the output state taken from the

"continuation" f into a list of scheduling-independent traces, for which compositionality can be achieved in both temporal and value domains, then *distribute* this state through the list of interleavings and finally combine the corresponding outputs using *reduce*.

The function *scatter* is trivially defined by the Haskell function *replicate*. This function returns a finite list which elements are copies of the input argument.

$$\begin{aligned} &scatter :: Int \rightarrow a \rightarrow [a] \\ &scatter = replicate \end{aligned}$$

The function *distribute* takes a list of functions $[a \rightarrow a]$, and a list of input values $[a]$ and return a list $[a]$ with the results obtained by applying each input function to each input value. This semantics is directly given by the Haskell function *zipWith*.

$$\begin{aligned} &distribute :: [a \rightarrow a] \rightarrow [a] \rightarrow [a] \\ &distribute = zipWith\ (\lambda f\ a \rightarrow f\ a) \end{aligned}$$

The function *reduce* is responsible for computing the least upper bound between the elements of the input list $[a]$. This function is applied only at those program points where processes synchronize, i.e. those program points where the abstract register and pipeline states of the different processing units are merged into a single abstract state, therefore releasing the resources allocated to the second processor core. To this end, we provide a proper instantiation of the *join* function for the new version of the datatype **CPU**.

$$\begin{aligned} &reduce :: (Lattice\ a) \Rightarrow [a] \rightarrow a \\ &reduce = foldl\ join\ bottom \end{aligned}$$

As already mentioned, each "architectural flow" interleaves instructions from different processes in a single trace. Therefore, the transition relations contained in each **Leaf** of a dependency graph can belong either to the main or the forked processes. Hence, we need to re-define the interpretation of *derive* so that the appropriate processing unit can be selected from the multicore hardware processor state. To this end, we define the functions *break* and *continue* of the type class *Synchronizable*, to allow us to specify which is the current "active" processor core and to select the appropriate corresponding pipelines.

$$derive\ f\ (\mathbf{Leaf}\ r) = f * (break\ r) * (refunct\ r) * (continue\ r)$$

The concrete timing property of hybrid pipeline states P of Def. (6.29) is used to keep track the analysis time elapsed during the intermediate analysis steps inside the pipelining of a single instruction, as well as the overall analysis time elapsed during the fixpoint computation. Along the lines of [117], the semantics of hybrid states can be viewed as an abstract semantics *instrumented* with a concrete timing property. This information about the analysis "time" allow us to compute the delays resulting from the shared requests by the fact that their arrival times are always known.

On one hand, the function *break* only affects the hardware state of a processor core, at the program point specified by the *label* of the input program *state*, if the input transition *relation*

belongs to the forked process. If that is the case, the function *breakPipelineAt* ensures that the active processing core has the identifier '*0*' (the main process) and that the analysis time taken from the processing core owning the input state is used to setup the starting time of the pipeline analysis of the "interleaved" instruction, which runs on the processor core with identifier '*1*'.

```
break :: Rel (St (CPU a)) → (St (CPU a)) → (St (CPU a))
break relation state@St {label, invs = i}
  = case isInterleaved relation of
      True → let at = point label
                 node = i ! at
                 cpu' = (breakPipelineAt ∘ value) node
                 node' = node {value = cpu'}
             in state {invs = insert at node' i}
      False → state

breakPipelineAt :: (Ord a) ⇒ CPU a → CPU a
breakPipelineAt cpu@CPU {active = 0}
  = (setSimTime cpu 1 (getSimTime cpu 0)) {active = 1}
```

On the other hand, the function *continue* defines how the main processor core regains control on the shared resources by setting the active processing core to '*0*'. Using the function *continuePipelineAt*, the analysis time at the end of an interleaved instruction is setup to be the starting time of the hardware state of the main processor core.

```
continue relation state@St {label, invs = i}
  = case isInterleaved r of
      True → let at = point label
                 node = i ! at
                 cpu' = (continuePipelineAt ∘ value) node
                 node' = node {value = cpu'}
             in state {invs = insert at node' i}
      False → state

continuePipelineAt :: (Cycles a) ⇒ CPU a → CPU a
continuePipelineAt cpu@CPU {active = 1}
  = (setSimTime cpu 0 (getSimTime cpu 1)) {active = 0}
```

The function *getSimTime* is used to obtain the concrete timing property associated to the "latest" pipeline state computed for a given processor core, and the function *setSimTime* is used to set the current analysis "time" of the pipeline analysis of the a given processor core.

```
getSimTime cpu@CPU {multi} target
  = let core@Core {pipeline = P♯ ps} = multi ! target
    in time $ maximum ps

setSimTime cpu@CPU {multi = m} target stamp
  = let core@Core {pipeline = P♯ ps} = m ! target
        pipeline' = P♯ (map (λp → p {time = stamp}) ps)
    in cpu {multi = insert target (core {pipeline = pipeline'}) m}
```

8.4 The $\mathcal{LR}$-server model as a Galois Connection

Assuming non-interference between processes in the value domain, i.e. composability in the value domain, and a generic arbitration protocol, possibly non-composable in the timing domain, we propose the use of the $\mathcal{LR}$-server model presented in [131] as an abstraction to achieve compositionallity in the pipeline analysis, so that the analysis of architectural flows can be avoided while preserving the soundness of timing analysis for multicore systems. The proof of soundness and the implementation of the $\mathcal{LR}$-server model in the context of data-flow analysis is supported by a Galois connection.

The meaning of the access times to shared resources in the context of timing analysis is the range of its possible values, i.e. the interval from lower bounds to upper bounds. Due to the limited bandwidth of the shared bus, shared accesses introduce additional delays that stall the pipeline. Therefore, the soundness of the timing analysis requires the computation of upper bounds on delays. To cope with this, *TimedTask* is redefined:

$$\mathit{TimedTask} \triangleq (\mathit{Cycles} \times \mathit{Delay} \times \mathit{Stage} \times \mathit{Task}) \tag{8.3}$$

As mentioned in Section 6.8, the pipeline abstract domain is defined as a set of hybrid pipeline states, each including a "concrete" timing property now given by *Cycles* plus *Delay*. The purpose of the $\mathcal{LR}$-server model is to reduce the number of joins and provide, at the same time, upper bounds for delays caused by shared requests. From the observation of Fig. 8.2(a), it is clear that the number of join operations is proportional to the number of architectural flows. However, Fig. 8.2(b) shows that when applying the $\mathcal{LR}$ model to compute *safe* upper bounds for the finishing times of shared requests, the number of joins is determined solely by the control flows of each process independently.

The soundness of the abstraction provided by the $\mathcal{LR}$-server model relies on the fact the all timing properties calculated throughout architectural flows are upper bounded by the finishing times calculated using the $\mathcal{LR}$ model. Here, the objective is to formalize this approximation by means of a Galois connection.

Let *Delay* be an upper semi-lattice equipped with a partial order $\leqslant$ on natural numbers $\mathbb{N}$, describing both concrete and abstract timing properties and let $\mathbb{D}$ be a set of timing properties. A Galois connection $\mathit{Delay}^{\natural}(\subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathit{Delay}^{\sharp}(\subseteq)$, where $\mathit{Delay}^{\natural} = \mathit{Delay}^{\sharp} = 2^{\mathbb{D}}$, is defined in terms of a *representation function* $\beta : \mathit{Delay} \mapsto \mathbb{D}$ that maps a concrete value $p \in \mathit{Delay}$ to the best property describing it in $\mathbb{D}$. This property is the canonical extension of Def. (8.2) to sets. Given a subset $X \subseteq \mathbb{D}$ and an abstract property $p^{\sharp} \in \mathit{Delay}^{\sharp}$, the abstraction and concretization maps are defined by:

$$\alpha(X) = \bigcup \{\beta(x) \mid x \in X\} \tag{8.4}$$

$$\gamma(p^{\sharp}) = \{p \in P \mid \beta(p) \subseteq p^{\sharp}\} \tag{8.5}$$

Let w_c^k be the k^{th} instruction to fetch from the shared memory when there is a cache miss in the processor core c. The best property $p^{\sharp}$ is the singleton set containing the smallest

finishing time given by Def. (8.2) when applied to w_c^k. Therefore, the $\mathcal{LR}$ abstraction can be formally defined by the representation function β:

$$\beta(t_f(w_c^k)) = \{\max(t_a(w_c^k) + \Theta_c, \hat{t}_f(w_c^{k-1})) + s(w_c^k)/\rho_c\} = \{\hat{t}_f(w_c^k)\} \tag{8.6}$$

This formally shows that the predictability of $\mathcal{LR}$ servers can be used to abstract the meta-programs corresponding to architectural flows into meta-programs corresponding to control flows only. Since each access time is upper bounded by the $\mathcal{LR}$ server, we have by compositionality that the maximum local timing property given by Def. (8.4), that would be obtained by joining ($\bigcup$) all abstract pipeline states across the architectural flows in Fig. 8.2(a), is exactly equal to the maximum local timing property when only the control flows are considered.

8.5 Haskell definitions for resource sharing

This section gives declarative definitions for the temporal behavior of TDM and $\mathcal{LR}$ arbiters. Let the polymorphic type variable a in (**CPU** a) be instantiated by a concrete timing property denoted by the data type **WCET**.

$$\textbf{data WCET} = \textbf{WCET}\ \{\mathit{cycles} :: \mathit{Int}, \mathit{arrival} :: \mathit{Int}, \mathit{core} :: \mathit{Int}, \mathit{finish} :: \mathit{Int}, \mathit{delay} :: \mathit{Int}\}$$

The analysis of a TDM arbiter is simplified due to its predictable and composable properties, which makes the delay of a request to a shared resource easily computed using the *arrival* time and the processor *core* identifier. As mentioned in Section 6.8, requests to the main instruction memory occur upon cache misses. Thus, the definition of the function *missed* is:

$$\begin{aligned}
&\mathit{missed}\ w@\textbf{WCET}\ \{\mathit{cycles} = c, \mathit{arrival}, \mathit{core}\}\\
&\quad = \textbf{let}\ d = \mathit{mod}\ \mathit{arrival}\ \mathit{frame}\\
&\qquad\quad \mathit{first} = \mathit{slots} * \mathit{core}\\
&\qquad\quad \mathit{end} = \mathit{first} + \mathit{slots} - 1\\
&\qquad\quad ts = \textbf{if}\ \mathit{first} \leqslant d \wedge d \leqslant \mathit{end}\ \textbf{then}\ 0\\
&\qquad\qquad\quad \textbf{else if}\ d < \mathit{first}\ \textbf{then}\ (\mathit{first} - d)\ \textbf{else}\ (\mathit{frame} - d + \mathit{first})\\
&\qquad \textbf{in}\ w\ \{\mathit{cycles} = c + \mathit{round}\ (ts + 1), \mathit{finish} = \mathit{arrival} + ts + 1, \mathit{delay} = ts + 1\}
\end{aligned}$$

The frame size of the TDM bus is given by the variable *frame*. Assuming slots are equally distributed among the processor cores and that they are consecutively allocated in the frame and a completion latency of 1 cycle, the delay time is $ts + 1$, where ts uses the division remainder of the *arrival* time by *frame* in order to check for an allocated slot. If the *core* needs to wait for an allocated slot, the required number of cycles can be statically calculated [71].

Now consider a shared bus with an arbitration protocol that is predictable but not composable, such as work-conserving round robin. In this case, the timing behavior of each application is dependent on the applications running on other cores, which makes analysis of all architectural flows mandatory in order to achieve soundness. In this context, the advantage of the $\mathcal{LR}$-server abstraction is the possibility to guarantee bounds on the starting

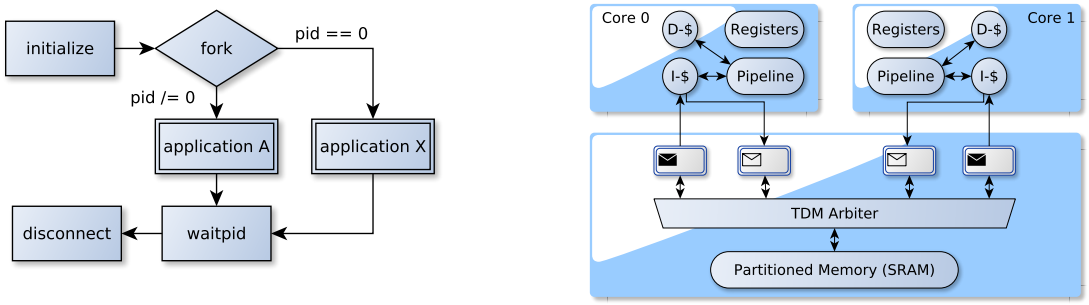
times and finishing times of the requests so that compositionality in the timing domain is achieved.

The $\mathcal{LR}$-server model requires a timing property to model the guaranteed service rate, which is the *finish* time of the previous request on the same *core*. According to Def. (8.2), the function *missed* defines the timing behavior of a cache miss in terms of an *arrival* time and a previous *finish* time. Therefore, the new definition of *missed* is:

$$
\begin{aligned}
&\mathit{missed}\ w@\mathbf{WCET}\ \{\mathit{cycles} = c, \mathit{arrival} = ta, \mathit{finish} = tf\} \\
&\quad = \mathbf{let}\ \mathit{busy} = \mathbf{if}\ ta + \mathit{theta} < tf \\
&\qquad\quad d = \mathbf{if}\ \mathit{busy}\ \mathbf{then}\ 1/\mathit{rho}\ \mathbf{else}\ \mathit{theta} + 1/\mathit{rho} \\
&\qquad \mathbf{in}\ w\ \{\mathit{cycles} = c + \mathit{round}\ d, \mathit{finish} = d + \mathbf{if}\ \mathit{busy}\ \mathbf{then}\ tf\ \mathbf{else}\ ta, \mathit{delay} = d\}
\end{aligned}
$$

8.6 Experimental Results

The discussion of experimental results include two different experimental scenarios. First, we compare the WCET and the analysis time obtained for small programs from the analysis of architectural flows (TDM) versus control flows (TDM) in Table 8.1. Second, we compare the WCET results of composable TDM versus a $\mathcal{LR}$ abstraction of a TDM arbiter for Mälardalen WCET benchmark programs [109] in Table 8.2. By compositionality of the $\mathcal{LR}$ abstraction and assuming that each processor core has a sufficiently large private data memory (D-$) and a common initial hardware state, each program is analyzed independently from the program configured to run on the second core. We consider the simplified multicore architecture in Fig. 8.4(b), where instructions are shared in a partitioned SRAM memory shared by a TDM arbiter.



(a) Template for a multi-process source program

(b) Simplified multi-processor architecture

Figure 8.4: A simple source code running on a simplified multicore architecture

By definition, architectural flows cannot be feasibly computed. However, we do compute interleavings for the simple program in Fig. 8.4(a), where "application A" and "application X" have only a few instructions each. Due to its natural composability, the analysis of control flows with TDM arbitration is much faster than the analysis of architectural flows, requiring

only 1% of the time. With respect to the WCET estimate, the first line in Table 8.1 shows a lower WCET (179 CPU cycles) for the interleavings approach compared to composable TDM analysis (185 CPU cycles). This difference in the WCET is a consequence of the actual hardware state of the processor core running "application X" upon the invocation of the *fork* procedure and demonstrates the impact that the intermediate hardware states have on the timing analysis of architectural flows.

In fact, when the number of instructions of "application X" is bigger than the number of instructions of "application A", the worst-case path corresponds to that of "application X". However, since the analysis of "application X" starts with an empty pipeline state, it naturally takes less CPU cycles to complete. After increasing the number of instructions in "application A", this effect is eliminated because the worst-case path becomes that of "application A". Consequently, for the two analyses, the WCET is equal in the last two experiments.

Table 8.1: Comparison results for architectural flows, composable TDM

No. instructions "application A"	No. instructions "application X"	No. of interleavings	Results (CPU cycles/sec.)	Architectural Flows (TDM)	Composable TDM
4	5	126	WCET	179	185
			Analysis Time	57.0	0.17
5	5	252	WCET	188	188
			Analysis Time	140.3	0.18
6	5	462	WCET	195	195
			Analysis Time	588.7	0.43

Next, we compare the WCET results in Table 8.2 obtained using the $\mathcal{LR}$ abstraction with $\Theta = 1$ and $\rho = 0.5$ (modeling a particular TDM configuration with frame size of 2) to the results obtained with composable TDM. The WCET values presented in Table 8.2 depend not only on the size of the instruction cache and on the ability of the $\mathcal{LR}$ server to stay busy, but also on the program flow, e.g. number of loop iterations. Since we are considering a blocking multicore architecture, where a request from a processor core cannot be issued before the previous request has been served, every request starts a new busy period by definition. This is the most unfavorable situation possible for the $\mathcal{LR}$ abstraction, since every request requires $\Theta + 1/\rho$ cycles to complete, maximizing the overhead compared to TDM.

Still, our experiments show that this overhead is limited to between 8.7% and 12.1% for the considered arbiter, configuration, and applications. This is partly because the use of a small frame size reduces the penalty of starting a new busy period upon every cache miss through the low $\Theta = 1$ value, but also because the case of an SRAM shared by a TDM arbiter is quite simple and is captured well by the abstraction. A more complex case with DRAM and CCSP arbitration is shown in [127] along with an optimization to reduce the pessimism of the abstraction without loss of generality. In terms of the run-time of the analysis tool, it is

approximately (≈) the same for both composable TDM and the $\mathcal{LR}$ abstraction.

From this experiment, we conclude that compositional analysis of control flows using the $\mathcal{LR}$ abstraction is very fast and scalable compared to analysis of architectural flows. The analysis time is similar to compositional analysis based on composable TDM arbitration, although it incurs a reduction in accuracy of about 8-12% for our configuration and applications. More precise WCET estimates would be obtained for multicore architectures that support high levels of parallelism. For example, architectures including super-scalar pipelines or caches allowing multiple outstanding requests. This would reduce the number of busy periods in the $\mathcal{LR}$ server upon cache misses, but would also increase the overall complexity of the WCET analyzer. Nevertheless, the main benefit of the $\mathcal{LR}$ abstraction is that it is able to perform compositional timing analysis using any arbiter belonging to the class, as opposed to being limited to composable TDM.

Table 8.2: WCET results for some of the Mälardalen benchmarks

Benchmark	No. Source Loop Iterations	$\mathcal{LR}$-server (WCET)	No. Cache Misses	TDM (WCET)	Overhead (%)	Analysis Time in sec. (≈)
bs	152	1162	111	1036	10.8	2.3
bsort	156	1459	152	1311	10.1	0.9
cnt	145	1309	175	1171	10.5	0.8
cover	111	796	105	707	11.2	3.9
crc	459	3160	304	2826	10.6	15.0
expint	251	2023	233	1818	10.1	1.9
fdct	1011	10897	720	9892	9.2	20.1
fibcall	111	994	59	885	11.0	2.3
matmult	287	2580	188	2343	9.2	5.2
minmax	221	956	263	873	8.7	2.6
prime	232	1079	196	959	11.1	5.2
ud	418	3943	97	3464	12.1	40.0

8.7 Summary

The work presented in this chapter is an approach to timing analysis in multicore architectures exclusively based on the declarative frameworks of denotational semantics, abstract interpretation and functional programming. Comparatively with the generic framework for data flow analysis described in Section 5, the WCET analysis in multicores is defined incrementally by extending the intermediate representation language with a new syntactic element, representing programs running on different processing cores, which denotational interpretation reuses the algebraic combinators used for static analysis in single-cores to automatically generate type-safe fixpoint (abstract)-interpreters.

The complexity of the new fixpoint interpreter is reduced by using the abstraction provided by the $\mathcal{LR}$ server model on the timing behavior of shared resources. This abstraction is

proved correct in relation to the calculational approach of "architectural flows" by means of a Galois connection. Using declarative programming in Haskell, the temporal behavior of shared resources is in direct correspondence with the mathematical definitions of the TDM and $\mathcal{LR}$ arbiter models. The outcome is the definition of provably sound and compositional timing analysis in multicore environments, with a loss in precision in order of 8% that is relatively small compared to the factor 100 reduction in terms of analysis time.

Chapter 9

Conclusion and Future Work

The main objective of the work reported in this dissertation is the definition of programming-language independent meta-semantic formalism, capable of specifying the fixpoint semantics of programs using typed and polymorphic higher-order combinators in Haskell. Sound and efficient fixpoint computations are obtained through the use of formal approaches to control-flow analysis combined with data-flow analysis within the same meta-semantic formalism.

The former is obtained by a type-safe fixpoint algorithm, automatically derived from a topological order over the syntactic elements of the program. The latter is obtained by applying a calculational method to the induction of "correct by construction" abstract interpreters. The meta-semantic formalism is defined to ease fixpoint verification and program transformations in the scenarios where the framework of Abstract-Carrying Code (ACC) can be applied.

The success of our approach is evaluated when the meta-semantic formalism is applied to the analysis of the *worst-case execution time* (WCET) of assembly programs, considering the ARM9 as the target platform. We show that the conservative approach to abstract interpretation proposed by the Cousots can be used to prove the correctness of the existent state-of-the-art on WCET analysis, in terms of the several static analyses required to compute a WCET estimate.

When using WCET safety specifications, the verification mechanism of the abstract interpretation part of ACC was extended with dual theory applied to linear programming (LP). In this way, the complexity of the LP problem on consumer sites is reduced from NP-hard to polynomial time, by using simple linear algebra computations. Therefore, we are able to provide an efficient and low-resource consuming verification mechanism.

Last but not least, we apply the *latency-rate* ($\mathcal{LR}$) server model to our WCET analysis with the objective to surpass the intrinsic computational complexity of timing analysis of multiple processing cores sharing common resources. The soundness of the integration of the $\mathcal{LR}$ timing abstraction into our data flow framework is proved using the abstract interpretation

framework based on Galois connections. Although the considered multicore architecture is rather simplified, the results show that the our solution for WCET analysis on multicores can be easily parametrized with an abstraction of the timing behavior of any arbiter for shared resources belonging to the class of $\mathcal{LR}$-servers.

9.1 Future Work

The two main limitations of our WCET analysis framework are the absence of the use of widening/narrowing operators to accelerate the convergence of fixpoint computations and the simplification of the real ARM9 cache replacement policies and hardware timing models. The first limitation is a consequence of the requirements imposed by the ACC framework, stating that the verification mechanism must be performed without manual intervention.

In fact, since program flow annotations on the source code are not allowed in ACC when performing static analysis of the machine code, we have to resort to complete loop unrolling to perform an automatic *program flow analysis* by abstract interpretation. This can be a considerably less efficient process when compared to existing state-of-the-art tools, such as AbsInt's `aiT`, but it produces more precise results by minimizing the non-determinism introduced by the separate use of different analyses.

The static analysis of a realistic ARM9 microprocessor depends greatly on its hardware components and may even be impossible to perform. For example, the cache replacement policy of ARM9 is typically Pseudo-Random. This replacement policy is highly unpredictable and precludes, to the best of our knowledge, the application of static analysis methods to determine approximations about the actual cache dynamic behavior. Indeed, state-of-the-art cache analysis consider either *Least Recently Used* LRU, *First-In-First-Out* FIFO or *Pseudo-LRU* PLRU [55]. For this reason, and for sake of simplicity, we restrict our calculational approach to abstract cache analysis using the LRU replacement policy, for which we give a correctness proof by construction.

Furthermore, recent published work on automatic generation of timing models from VHDL microprocessor specifications [118], would allow the automatically generation of Haskell code to include the pipeline timing model of ARM9. Although some progress was made in this direction in cooperation with AbsInt GmbH [2] and in cooperation with the Compiler Lab Design at Saarbrücken University, the results of such work are not mature enough and are, therefore, outside the scope of this thesis. Of course, the absence of a realistic timing model for ARM9 will influence the uniformity of our WCET estimates when compared to AbsInt's tool, for example.

9.2 Final Considerations

An important objective of this thesis is directly related to use of Haskell to prove the correctness of type specifications and to use the formalism of denotational semantics in the calculational process of inducing abstract interpreters that are "correct by construction".

Indeed, the polymorphic type system of Haskell allows us to define a polymorphic two-level meta-language and a parametrized fixpoint semantics for free. Moreover, the compositional aspect of the analyzer is trivially implemented by functional composition. Finally, the Haskell definitions of the static analyzer, being highly declarative, are a direct implementation of the corresponding denotational definitions we have obtained by calculus on paper.

In summary, we have shown that Haskell can be used as a language in which the mathematical complex notions underlying the theory of abstract interpretation can be easily, elegantly, and efficiently implemented and applied to the analysis of complex hardware architectures. Additionally, we have shown that the declarative paradigm of Haskell is extremely useful and synthetic when expressing specific and complex mathematical definitions by means of embedded domain specific languages and a direct and simple programming paradigm to express analytical formalisms.

Bibliography

[1] Samson Abramsky and Chris Hankin, editors. *Abstract Interpretation of Declarative Languages*. Ellis Horwood, 1987.

[2] AbsInt. Angewandte informatik. http://www.absint.com/pag/.

[3] B. Ackland et al. A single-chip 1.6 billion 16-b mac/s multiprocessor dsp. *IEEE Journal of Solid-state Circuits*, 35(3):412–424, 2000.

[4] Mads Sig Ager et al. A functional correspondence between call-by-need evaluators and lazy abstract machines. *Inf. Process. Lett.*, 90(5):223–232, June 2004.

[5] Joaquín Aguado and Michael Mendler. Computing with streams. In *Proc. of the sixth workshop on Declarative aspects of multicore programming*, DAMP '11, pages 35–44, New York, NY, USA, 2011. ACM.

[6] Benny Akesson et al. Composable resource sharing based on latency-rate servers. In *DSD*, pages 547–555, 2009.

[7] Benny Akesson and Kees Goossens. Architectures and modeling of predictable memory controllers for improved system integration. In *DATE*, pages 851–856, 2011.

[8] Hussein Al-Zoubi et al. Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite. In *Proc. of the 42nd annual Southeast regional conference*, ACM-SE 42, pages 267–272, New York, NY, USA, 2004. ACM.

[9] Elvira Albert et al. Abstraction-carrying code. In *LPAR*, pages 380–397, 2004.

[10] Elvira Albert et al. An abstract interpretation-based approach to mobile code safety. *Electron. Notes Theor. Comput. Sci.*, 132(1):113–129, 2005.

[11] Elvira Albert et al. Certificate size reduction in abstraction-carrying code. *CoRR*, abs/1010.4533, 2010.

[12] Frances Allen et al. The experimental compiling system. *IBM Journal of Research and Development*, 24:695–715, 1980.

[13] Lloyd Allison. *A practical introduction to denotational semantics.* Cambridge University Press, New York, NY, USA, 1986.

[14] Alexandru Andrei et al. Predictable implementation of real-time applications on multiprocessor systems-on-chip. In *Proc. of the 21st International Conference on VLSI Design*, VLSID '08, pages 103–110, Washington, DC, USA, 2008. IEEE Computer Society.

[15] Kevin Backhouse and Roland Backhouse. Logical relations and galois connections. In *MPC '02: Proc. of the 6th International Conference on Mathematics of Program Construction*, pages 23–39, London, UK, 2002. Springer-Verlag.

[16] John Backus. Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, August 1978.

[17] Gilles Barthe et al. Mobius: mobility, ubiquity, security objectives and progress report. In *Proc. of the 2nd international conference on Trustworthy global computing*, TGC'06, pages 10–29, Berlin, Heidelberg, 2007. Springer-Verlag.

[18] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Syst. J.*, 5(2):78–101, June 1966.

[19] Frédéric Besson et al. Certified static analysis by abstract interpretation. In *Foundations of Security Analysis and Design V: FOSAD 2007/2008/2009 Tutorial Lectures*, pages 223–257, Berlin, Heidelberg, 2009. Springer-Verlag.

[20] François Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proc. of the International Conference on Formal Methods in Programming and their Applications*, pages 128–141. Springer-Verlag, 1993.

[21] Marius Bozga et al. Kronos: A model-checking tool for real-time systems. In *CAV*, pages 546–550, 1998.

[22] Bernd Brassel and Jan Christiansen. Towards a new denotational semantics for curry and the algebra of curry. Technical report, Christian-Albrechts-Universitat Kiel, 2007.

[23] David Cachera and David Pichardie. A certified denotational abstract interpreter. In *Proc.. of International Conference on Interactive Theorem Proving (ITP-10)*, Lecture Notes in Computer Science. Springer-Verlag, 2010. To appear.

[24] Sudipta Chattopadhyay et al. Modeling shared cache and bus in multi-cores for timing analysis. In *Proc. of the 13th International Workshop on Software & Compilers for Embedded Systems*, SCOPES '10, pages 6:1–6:10, New York, NY, USA, 2010. ACM.

[25] Patrick Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes (in French).* Thèse d'État ès sciences mathématiques, Université Joseph Fourier, Grenoble, France, 21 March 1978.

[26] Patrick Cousot. Semantic foundations of program analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[27] Patrick Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Electronic Notes in Theoretical Computer Science*, 6, 1997.

[28] Patrick Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design.* NATO ASI Series F. IOS Press, Amsterdam, 1999.

[29] Patrick Cousot. Partial completeness of abstract fixpoint checking, invited paper. In *Proc. of the Fourth International Symposium on Abstraction, Reformulations and Approximation, SARA'2000, Lecture Notes in Artificial Intelligence 1864*, pages 1–25, Horseshoe Bay, Texas, USA, 26–29 July 2000. Springer-Verlag, Berlin, Germany.

[30] Patrick Cousot. Abstract interpretation based formal methods and future challenges, invited paper. In R. Wilhelm, editor, *¡¡ Informatics — 10 Years Back, 10 Years Ahead ¿¿*, volume 2000 of *Lecture Notes in Computer Science*, pages 138–156. Springer-Verlag, 2001.

[31] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *Proc. of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.

[32] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[33] Patrick Cousot and Radhia Cousot. Automatic synthesis of optimal invariant assertions: Mathematical foundations. *SIGPLAN Not.*, 12(8):1–12, August 1977.

[34] Patrick Cousot and Radhia Cousot. Constructive versions of Tarski's fixed point theorems. *Pacific Journal of Mathematics*, 81(1):43–57, 1979.

[35] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.

[36] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992.

[37] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2:511–547, 1992.

[38] Patrick Cousot and Radhia Cousot. Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In *PLILP '92: Proc. of the 4th International Symposium on Programming Language Implementation and Logic Programming*, pages 269–295, London, UK, 1992. Springer-Verlag.

[39] Patrick Cousot et al. The ASTRÉE analyzer. In *Programming Languages and Systems, Proc. of the 14th European Symposium on Programming, volume 3444 of Lecture Notes in Computer Science*, pages 21–30. Springer, 2005.

[40] Christoph Cullmann et al. Predictability considerations in the design of multi-core embedded systems. *Ingénieurs de l'Automobile*, 807:36–42, September 2010.

[41] Olivier Danvy and Kevin Millikin. Refunctionalization at work. *Sci. Comput. Program.*, 74(8):534–549, June 2009.

[42] B. A. Davey and H. A. Priestly. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[43] Jakob Engblom and Andreas Ermedahl. Modeling complex flows for worst-case execution time analysis. In *Proc. of the 21st IEEE conference on Real-time systems symposium*, RTSS'10, pages 163–174, Washington, DC, USA, 2000. IEEE Computer Society.

[44] Andreas Ermedahl and Jan Gustafsson. Deriving annotations for tight calculation of execution time. In *Proc. of the Third International Euro-Par Conference on Parallel Processing*, Euro-Par '97, pages 1298–1307, London, UK, 1997. Springer-Verlag.

[45] Andreas. Ermedahl and Mikael Sjödin. Interval analysis of c-variables using abstract interpretation. Technical report, Uppsala University, 1996.

[46] Heiko. Falk et al. Design of a wcet-aware c compiler. In *Proc. of the 2006 IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia*, ESTMED '06, pages 121–126, Washington, DC, USA, 2006. IEEE Computer Society.

[47] Heiko Falk et al. Compile-time decided instruction cache locking using worst-case execution paths. In *Proc. of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '07, pages 143–148, New York, NY, USA, 2007. ACM.

[48] Christian Ferdinand et al. Cache behavior prediction by abstract interpretation. *Sci. Comput. Program.*, 35:163–189, November 1999.

[49] Christian Ferdinand and Reinhold Heckmann. ait: Worst-case execution time prediction by static program analysis. In Renè Jacquart, editor, *Building the Information Society*, volume 156 of *IFIP International Federation for Information Processing*, pages 377–383. Springer Boston, 2004.

[50] Christian Ferdinand and Reinhard Wilhelm. On predicting data cache behavior for real-time systems. In *Proc. of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, LCTES '98, pages 16–30, London, UK, UK, 1998. Springer-Verlag.

[51] Mohammad Ali Ghodrat et al. Control flow optimization in loops using interval analysis. In *CASES '08: Proc. of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, pages 157–166, New York, NY, USA, 2008. ACM.

[52] Jeremy Gibbons. A pointless derivation of radixsort. *Journal of Functional Programming*, 9(3):339–346, 1999.

[53] GLPK. http://www.gnu.org/software/glpk.

[54] Michael J. C. Gordon. *The Denotational Description of Programming Languages: An Introduction.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1979.

[55] Daniel Grund. *Static Cache Analysis for Real-Time Systems – LRU, FIFO, PLRU.* PhD thesis, Saarland University, 2012.

[56] Daniel Grund and Jan Reineke. Abstract interpretation of fifo replacement. In *Proc. of the 16th International Symposium on Static Analysis*, SAS '09, pages 120–136, Berlin, Heidelberg, 2009. Springer-Verlag.

[57] Jan Gustafsson and Andreas Ermedahl. Automatic derivation of path and loop annotations in object-oriented real-time programs. In Lonni R. Welch and Dieter K. Hammer, editors, *Engineering of distributed control systems*, pages 81–98. Nova Science Publishers, Inc., Commack, NY, USA, 2001.

[58] Andreas Hansson et al. Enabling application-level performance guarantees in network-based systems on chip by applying dataflow analysis. *IET Computers & Digital Techniques*, 3(5):398–412, 2009.

[59] D. Hardy and I. Puaut. Wcet analysis of multi-level non-inclusive set-associative instruction caches. In *Proc. of the 29th Real-Time Systems Symposium*, pages 456–466, Barcelona, Spain, December 2008.

[60] Damien Hardy et al. Using bypass to tighten wcet estimates for multi-core processors with shared instruction caches. In *Proc. of the 2009 30th IEEE Real-Time Systems Symposium*, RTSS '09, pages 68–77, Washington, DC, USA, 2009. IEEE Computer Society.

[61] Christopher Healy et al. Bounding loop iterations for timing analysis. In *Proc. of the Fourth IEEE Real-Time Technology and Applications Symposium*, RTAS '98, pages 12–, Washington, DC, USA, 1998. IEEE Computer Society.

[62] Christopher Healy et al. Supporting timing analysis by automatic bounding of loop iterations. *Real-Time Systems*, 18(2/3):129–156, May 2000.

[63] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach.* The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2011.

[64] Manuel Hermenegildo et al. Abstraction carrying code and resource-awareness. In *Proc. of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '05, pages 1–11, New York, NY, USA, 2005. ACM.

[65] T. Hickey, Q. Ju, and M. H. Van Emden. Interval arithmetic: From principles to implementation. *J. ACM*, 48(5):1038–1068, September 2001.

[66] Frederick S. Hillier and Gerald J. Lieberman. *Introduction to operations research, 4th ed.* Holden-Day, Inc., San Francisco, CA, USA, 1986.

[67] A. Hoffman and J. Kruskal. Integral boundary points of convex polyhedra, in Linear Inequalities and Related Systems (H. Kuhn and A. Tucker, Eds.). *Annals of Maths. Study*, 38:223–246, 1956.

[68] Gerard Holzmann. *Spin model checker, the: primer and reference manual.* Addison-Wesley Professional, first edition, 2003.

[69] Neil Jones and Alan Mycroft. Data flow analysis of applicative programs using minimal function graphs. In *Proc. of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '86, pages 296–306, New York, NY, USA, 1986. ACM.

[70] Neil Jones and Flemming Nielson. Abstract interpretation: a semantics-based tool for program analysis. In *Handbook of logic in computer science (vol. 4): semantic modelling*, pages 527–636, Oxford, UK, 1995. Oxford University Press.

[71] Timon Kelter et al. Bus-aware multicore wcet analysis through tdma offset bounds. In *Proceedings of the 2011 23rd Euromicro Conference on Real-Time Systems (ECRTS)*, pages 3–12, 2011.

[72] Gary Kildall. A unified approach to global program optimization. In *Proc.. of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL'73, pages 194–206, New York, NY, USA, 1973. ACM.

[73] Matz Kindahl. The galois connection in interval analysis. Docs, Uppsala University, Sweden, August 1996.

[74] Stephen Cole Kleene. *Introduction to metamathematics*. Van Nostrand, 1952.

[75] P. Lacan et al. ARIANE 5 - The Software Reliability Verification Process. In *DASIA 98 - Data Systems in Aerospace*, volume 422 of *ESA Special Publication*, May 1998.

[76] Kim G. Larsen et al. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1:134–152, 1997.

[77] P. Lee and U. Pleban. A realistic compiler generator based on high-level semantics: another progress report. In *Proc. of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '87, pages 284–295, New York, NY, USA, 1987. ACM.

[78] Yan Li et al. Timing analysis of concurrent programs running on shared cache multi-cores. In *IEEE Real-Time Systems Symposium*, pages 57–67, 2009.

[79] Yau-Tsun Steven Li et al. Cache modeling for real-time software: beyond direct mapped instruction caches. In *IEEE Real-Time Systems Symposium*, pages 254–263, 1996.

[80] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Proc. of the 32nd annual ACM/IEEE Design Automation Conference*, DAC '95, pages 456–461, New York, NY, USA, 1995. ACM.

[81] Sung-Soo Lim et al. An accurate worst case timing analysis for risc processors. *IEEE Trans. Softw. Eng.*, 21(7):593–604, July 1995.

[82] Paul Lokuciejewski and Peter Marwedel. Combining worst-case timing models, loop unrolling, and static loop analysis for wcet minimization. In *Proc. of the 2009 21st Euromicro Conference on Real-Time Systems*, pages 35–44, Washington, DC, USA, 2009. IEEE Computer Society.

[83] Thomas Lundqvist and Per Stenström. Integrating path and timing analysis using instruction-level simulation techniques. In *Proc. of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, LCTES '98, pages 1–15, London, UK, UK, 1998. Springer-Verlag.

[84] Thomas Lundqvist and Per Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems*, 17(2-3):183–207, December 1999.

[85] Zohar Manna. *Mathematical Theory of Computation.* Dover Publications, Incorporated, 2003.

[86] Florian Martin et al. Analysis of loops. In *CC*, pages 80–94, 1998.

[87] J. Mccarthy. Towards a mathematical science of computation. In *In IFIP Congress*, pages 21–28. North-Holland, 1962.

[88] Ross M. McConnell et al. Certifying algorithms. *Computer Science Review*, 5(2):119–161, 2011.

[89] Matthew Might. Abstract interpreters for free. In *Proc. of the 17th international conference on Static analysis*, SAS'10, pages 407–421, Berlin, Heidelberg, 2010. Springer-Verlag.

[90] Richard Mitchell, Jim McKim, and Bertrand Meyer. *Design by contract, by example.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2002.

[91] Greg Morrisett et al. From system f to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21:527–568, May 1999.

[92] Frank Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2/3):217–247, May 2000.

[93] George C. Necula. Proof-carrying code. In *Proc. of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '97, pages 106–119, New York, NY, USA, 1997. ACM.

[94] Flemming Nielson. A denotational framework for data flow analysis. *Acta Informatica*, 18:265–287, 1982.

[95] Flemming Nielson. *Abstract Interpretation Using Domain Theory.* PhD thesis, University of Edinburgh, 1984.

[96] Flemming Nielson. Abstract interpretation of denotational definitions. In *3rd annual symposium on theoretical aspects of computer science on STACS 86*, pages 1–20, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[97] Flemming Nielson. Program transformations in a denotational setting. *ACM Trans. Program. Lang. Syst.*, 7(3):359–379, July 1985.

[98] Flemming Nielson et al. *Principles of Program Analysis.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[99] Flemming Nielson and Hanna. R. Nielson. Code generation from two-level denotational meta-languages. In *on Programs as data objects*, pages 192–205, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[100] Hanne Riis Nielson and Flemming Nielson. Pragmatic aspects of two-level denotational meta-languages. In *Proc. of the European Symposium on Programming*, ESOP '86, pages 133–143, London, UK, 1986. Springer-Verlag.

[101] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer (Undergraduate Topics in Computer Science)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[102] Greger Ottosson and Mikael Sjodin. Worst-case execution time analysis for modern hardware architectures. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97)*, pages 47–55, 1997.

[103] Sascha Plazar et al. A Retargetable Framework for Multi-objective WCET-aware High-level Compiler Optimizations. In *Proc. of The 29th IEEE Real-Time Systems Symposium (RTSS)*, pages 49–52, Barcelona / Spain, December 2008.

[104] P. Puschner and Ch. Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1:159–176, 1989. 10.1007/BF00571421.

[105] Peter P. Puschner and Anton V. Schedl. Computing maximum task execution times - a graph-based approach. *Real-Time Systems*, 13(1):67–91, July 1997.

[106] Jan Reineke. *Caches in WCET Analysis: Predictability - Competitiveness - Sensitivity*. PhD thesis, Saarland University, 2009.

[107] Jan Reineke et al. A definition and classification of timing anomalies. In *6th Intl Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 1–6, 2006.

[108] Jan Reineke et al. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, August 2007.

[109] Mälardalen WCET research group. www.mrtc.mdh.se/projects/wcet.

[110] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proc. of the ACM annual conference - Volume 2*, ACM '72, pages 717–740, New York, NY, USA, 1972. ACM.

[111] Xavier Rival. Abstract interpretation-based certification of assembly code. In *VMCAI 2003: Proc. of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 41–55, London, UK, 2003. Springer-Verlag.

[112] Vítor Rodrigues, Benny Akesson, Simão Melo de Sousa, and Mário Florido. A declarative compositional timing analysis for multicores using the latency-rate abstraction. In *Fifteenth International Symposium on Practical Aspects of Declarative Languages (PADL)*, 2013. To appear.

[113] Vítor Rodrigues, João Pedro Pedroso, Mário Florido, and Simão Melo de Sousa. Certifying execution time. In *Proc. of the Second international conference on Foundational and Practical Aspects of Resource Analysis*, FOPARA'11, pages 108–125, Berlin, Heidelberg, 2012. Springer-Verlag.

[114] Vítor Rodrigues, Mário Florido, and Simão Melo de Sousa. Back annotation in action: from wcet analysis to source code verification. In *Actas of CoRTA 2011: Compilers, Prog. Languages, Related Technologies and Applications*, pages 276 – 281, July 2011.

[115] Vítor Rodrigues, Mário Florido, and Simão Melo de Sousa. A functional approach to worst-case execution time analysis. In *20th International Workshop on Functional and (Constraint) Logic Programming (WFLP)*, July 2011.

[116] Jakob Rosen et al. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *Proc. of the 28th IEEE International Real-Time Systems Symposium*, RTSS '07, pages 49–60, Washington, DC, USA, 2007. IEEE Computer Society.

[117] Mads Rosendahl. *Abstract Interpretation and Attribute Grammars.* PhD thesis, Cambridge University, 1991.

[118] Marc Schlickling and Markus Pister. Semi-automatic derivation of timing models for WCET analysis. In *LCTES '10: Proc. of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems*, pages 67–76. ACM, April 2010.

[119] Simon Schliecker et al. Integrated analysis of communicating tasks in mpsocs. In *Proc. of the 4th international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '06, pages 288–293, New York, NY, USA, 2006. ACM.

[120] David A. Schmidt. Abstract interpretation from a denotational-semantics perspective. *Electron. Notes Theor. Comput. Sci.*, 249:19–37, 2009.

[121] G. Schmidt and T. Ströhlein. *Relations and graphs: discrete mathematics for computer scientists.* EATCS monographs on theoretical computer science. Springer-Verlag, 1993.

[122] Jörn Schneider and Christian Ferdinand. Pipeline behavior prediction for superscalar processors by abstract interpretation. *SIGPLAN Not.*, 34:35–44, May 1999.

[123] Martin Schoeberl et al. A single-path chip-multiprocessor system. In *Proc. of the 7th IFIP WG 10.2 International Workshop on Software Technologies for Embedded and Ubiquitous Systems*, SEUS '09, pages 47–57, Berlin, Heidelberg, 2009. Springer-Verlag.

[124] Michael I. Schwartzbach. Lecture notes on static analysis. University of Aarhus, 2008.

[125] Simon Segars et al. The arm9 family - high performance microprocessors for embedded applications. In *Proc. of the International Conference on Computer Design*, pages 230–235. ARM Ltd, 1998.

[126] Ravi Sethi. Control flow aspects of semantics-directed compiling. *ACM Trans. Program. Lang. Syst.*, 5(4):554–595, October 1983.

[127] Hardik Shah, Alois Knoll, and Benny Akesson. Bounding SDRAM Interference: Detailed Analysis vs. Latency-Rate Analysis. In *Proc. DATE (to appear)*, 2013.

[128] Micha Sharir and Amir Pnueli. *Two Approaches to Interprocedural Data Flow Analysis*, pages 189–233. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1981.

[129] M. B. Smyth and G. D. Plotkin. The category-theoretic solution of recursive domain equations. In *Proc. of the 18th Annual Symposium on Foundations of Computer Science*, pages 13–17, Washington, DC, USA, 1977. IEEE Computer Society.

[130] Ingmar Jendrik Stein. *ILP-based path analysis on abstract pipeline state graphs.* PhD thesis, Saarlandische Universitäts- und Landesbibliothek, Postfach 151141, 66041 Saarbrücken, 2010.

[131] Dimitrios Stiliadis and Anujan Varma. Latency-rate servers: a general model for analysis of traffic scheduling algorithms. *IEEE/ACM T. Netw.*, 6(5):611–624, 1998.

[132] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, MA, USA, 1977.

[133] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[134] The DWARF Debugging Standard . http://www.dwarfstd.org/.

[135] The GraphML File Format. http://graphml.graphdrawing.org/.

[136] H. Theiling and C. Ferdinand. Combining abstract interpretation and ilp for microarchitecture modelling and program path analysis. In *Proc. of the IEEE Real-Time Systems Symposium*, RTSS '98, pages 144–, Washington, DC, USA, 1998. IEEE Computer Society.

[137] Henrik Theiling et al. Fast and precise wcet prediction by separated cache and path analyses. *Real-Time Systems*, 18(2/3):157–179, May 2000.

[138] Stephan Thesing. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland University, Germany, July 2004.

[139] Jelte Peter Vink et al. Performance analysis of soc architectures based on latency-rate servers. In *Proc. of the conference on Design, automation and test in Europe*, DATE '08, pages 200–205, New York, NY, USA, 2008. ACM.

[140] Vítor Rodrigues. Haskell prototype of a WCET static analyzer, 2012. available from: http://www.dcc.fc.up.pt/~vitor.rodrigues/.

[141] Maarten Wiggers et al. Modelling run-time arbitration by latency-rate servers in dataflow graphs. In *Proc.. SCOPES*, 2007.

[142] Reinhard Wilhelm. Why ai + ilp is good for wcet, but mc is not, nor ilp alone. In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*, pages 309–322. Springer Berlin / Heidelberg, 2003.

[143] Reinhard Wilhelm et al. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 28(7):966–978, July 2009.

[144] Reinhard Wilhelm and Björn Wachter. Abstract interpretation with applications to timing validation. In *CAV '08: Proc. of the 20th international conference on Computer Aided Verification*, pages 22–36, Berlin, Heidelberg, 2008. Springer-Verlag.

[145] Ying Yi et al. An ilp formulation for task mapping and scheduling on multi-core architectures. In *Proc. of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 33–38, 3001 Leuven, Belgium, 2009. European Design and Automation Association.

[146] Shuchang Zhou. An efficient simulation algorithm for cache of random replacement policy. In *Proc. of the 2010 IFIP international conference on Network and parallel computing*, NPC'10, pages 144–154, Berlin, Heidelberg, 2010. Springer-Verlag.